INPUT AND OUTPUT FACILITIES IN PL 7044

A Thesis Submitted
In Partial Fulfilment of the Requirements
For the Degree of

MASTER OF TECHNOLOGY

by

KRISHNAMURTHI KANNAN

to the
Department of Electrical Engineering
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

August, 1972

CERTIFICATE

Certified that the work " Input and Output Facilities in PL 7044" submitted by K. Kannan has been carried out under my supervision and that it has not been submitted elsewhere for a degree.

H.V. Sahasrabuddhe
(H.V. Sahasrabuddhe)
Assistant Professor
Electrical Engineering Dept.
Indian Institute of Technology
Kanpur

M-KAN-INP.

Thesis
001.64124
K133

To

The System IBM 7044

On which

I learnt the rudiments

Of programming

ACKNOWLEDGEMENTS

It is my pleasant duty to record here my deep sense of gratitude to Dr. Hari V. Sahasrabuddhe, Assistant Professor, Department of Electrical Engineering, firstly for suggesting this challenging project and secondly for the many enlightening discussions we have had during the course of this work. His constructive criticism and suggestions were always readily available to me and this has been a source of strength throughout.

My colleagues in the project, Mr. Ashok Gupta and Mr. A.B. Saha have displayed excellent team spirit, and have extended their whole hearted cooperation to me during my association with them. I shall forever cherish their companionship.

The Computer Centre staff have been obliging enough to have let me try out my ideas on the IBM 7044.

Mr. J.K. Misra's impeccable typing has helped me in no small measure in compiling this report in a short time.

Finally, I thank all my friends in the Campus for making my stay here a memorable one.

K. Kannan

IIT Kanpur-16

August, 1972

PREFACE

This thesis and two other reports, namely 'Program and Data Structures in PL 7044' and 'An Expression Processor for PL 7044' describe in detail a compiler writing project for the language PL/I on system IBM 7044. It has been the endeavour of the authors to present as much information about the various aspects of the project such as formulation, coding debugging methods etc. as is possible in a report of this nature.

We have tried to cater basically to two different types of readers, a) the casual reader who is interested in a bird's eye view of the project but not in details and b) the serious reader who is planning to embark on a similar project. It is in satisfying the demands of the latter category of readers that this report may be found inadequate by at least some of them, because it is extremely difficult to recognise when one has given sufficient details about a particular strategy adopted and at the same time ensure that no redundancy has crept in. Also since the project has been described in three separate volumes, which are not necessarily logically independent, a certain amount of incoherency may be discernable. We have tried to solve the first problem by illustrating all important aspects by suitable examples and by showing how the compiler goes about decoding them into MAP instructions. The second problem has been partially overcome by including a large number of Appendices at the end of each of three reports.

To enumerate systematically, Chapter I is an introduction to the language itself. We have obviously refrained from cataloguing all the features of the language, for this will entail a book by itself; we have instead highlighted those features in the language that are responsible for its enhanced status. PL 7044, which is a fairly large subset of PL/I is described in some detail in Chapter II. Here the emphasis is on the selection of a subset, the factors that influenced us in the selection of and some interesting techniques adopted to implement some of the powerful features incorporated in PL 7044. Chapter III deals with the lexical analyser and associated routines while Chapter IV carries a description of the I/O statement analyser and the DO statement analyser. Chapter V describes the Format Translator while Chapter VI gives details of Run Time Routines written for supporting PL 7044 I/O.

IBM's PL/I Language Specifications manual, Order No. GY33-6003-2, is the parent,source of the Syntax Notation, General Format, Syntax Rules and General Rules for various statements.

At the time of writing this report, coding and debugging of individual routines comprising the compiler has been completed, although test running the routines together could not be carried out due to unavoidable circumstances.

CONTENTS

LIST OF APPENDICES

SYNTAX NOTATION USED

1. Braces $\{\}$ are used, to denote grouping. Both vertical stacking of syntactical units and vertical stroke have been used to indicate that a choice is to be made e.g. both

$$\text{identifier} \begin{Bmatrix} \text{FIXED} \\ \text{FLOAT} \end{Bmatrix}$$

and identifier $\{\text{FIXED}|\text{FLOAT}\}$

have same meaning, i.e. 'identifier' must be followed by the literal occurrance of either the word FIXED or the word FLOAT but not both.

2. Square brackets $[\;]$ denote options. Anything enclosed in brackets may appear once or may not appear at all.

3. Three dots ... denote the occurrance of immediately preceding syntactical unit one or more times in succession.

4. Lower case letters are used to show nonterminal (syntactic) variables where as only CAPITAL letters are used to construct a terminal variable.

CHAPTER I

OVERVIEW OF THE PL/I LANGUAGE

1. PL/I: An Introduction:

1.1 Why PL/I:

The rapid growth of the computing machinery during the past two decades has necessiated a constant re-evaluation of the means by which the digital computer could be put to efficient use in the service of mankind. This is especially true in the field of language development. A consequence of this development has been PL/I - a language designed and promoted by IBM. Today PL/I is a fairly well established language and is fast gaining currency among users of IBM as well non-IBM equipment. Chief among the reasons for its success has been the fact that IBM have gone to great lengths in making the language versatile and useful to a wide range of computer users and at the same time making each class of users feel that the language has been tailored to its needs. This is not to say that PL/I has no shortcomings. To date, PL/I has not been implemented in toto on any digital machine and this is reflective of the practical difficulties in writing efficient compilers for PL/I. However, fairly large subsets have been introduced on IBM machines and whatever may be ones reservations about it, one has to learn to use PL/I if one wishes to take advantage of the large number of scientific and commercial programs written in PL/I. A logical extension of this necessity is the availability of a PL/I compiler within ones reach. To provide such a compiler to the users of IBM 7044 has been the goal of Project PL/7044.

1.2 PL/I: Its Philosophy:

In order to appreciate how PL/I is different from other high level languages it is necessary to examine the goals of the designers. These were a) to serve the needs of an unusually wide spectrum of users - scientific, commercial, real time and systems programmers - and to allow both the novice and the expert to find facilities at his own level b) to design the language such that programs could be written in accordance with the natural description of the problem it is supposed to solve so as to minimise coding errors and c) to provide a language for implementing present and future generations of compilers, monitors and the like - the question here being not so much of 'Can we write PL/I in PL/I' (we certainly can) but of 'Can we write in PL/I a real time operating system to support PL/I program'.

Not all of the above goals have been achieved in PL/I nor can it be said that any one of them has been completely achieved, but it does equip the user with tools that are not available in any other high level language.

In aiming for the goals enunciated above, certain principles were laid down which formed the framework of the new language and every feature of the language was motivated by one or more of these principles.

These principles were

(i) Anything goes: The new language PL/I should have a liberal semantic structure. If a reasonable interpretation can be given to a particular combination of symbols then it should be allowed per se by the language. A PL/I compiler should not have too many permissive diagnostics (" This is wrong but I am smart enough to know what you really mean"), on the other hand, it should have warning messages (" Do you really want this to be done in this strange way"). Such messages will enable compatibility among different implementations.

(ii) Full access to machine and system facility: This is a particularly useful guidline in so far as systems programmers are concerned but very difficult to achieve in practice.

(iii) Modularity: 'If you need it, it is there, but if you don't, you don't have to know about its existence". This was the motivation behind giving every name, every option, every facility a default interpretation which is so chosen that it is likely to be the one required by the user who is unaware that alternatives exist. However, one has to pay a penalty for this kind of a design, for if one forgets to assign an attribute other than the default attribute, no diagnostics will be provided.

(iv) Relative machine independence: Although this is a sound principle in itself, it is also one that most formulators tend to overlook. The makers of the PL/I language have achieved a measure of success in this area but even then PL/I is not altogether free of this shortcoming. In fact it does tend to reflect the organisation of the IBM 360 family of computers, particularly in the field of data structures.

(v) Catering to the novice: More than one method of specifying a particular problem is made available to cater to the needs of the novice. This will enable the user to use the notations most familiar to him and at the same time allow the compiler to maximise efficiency of the most frequently used case.

(vi) Choice of PL/I formats: A free format has been chosen to facilitate ease of coding and punching. 48 and 60 character sets are provided so as to allow implementation on different machines.

2. PL/I: Some Features:

For obvious reasons, it is neither desirable nor feasible to catalogue the large repertoire of PL/I features that have distinguished the language from other high level languages. However, we shall attempt to highlight a few salient features that are mainly responsible for its enhanced status.

2.1 Program Structure:

2.1.1 Basic Language Elements:

As already mentioned PL/I allows the program to be written in a free field format. The basic element is the statement which is separated from another statement by the semicolon and is a combination of characters (from either the 48 or 60 character set). Thus a program is simply a string of characters with no internal grouping.

2.1.2 Groups:

An important aspect of PL/I program structure is the entity called GROUP. Quite often, in programming, we are faced with a decision structure where the alternatives require the execution of more than one statement. In FORTRAN, either we use a subroutine or branch off to some other part of the program and then come back to the point from where we branched off. Both these methods are unsatisfactory the first because one may have to prepare the subroutine in the sense of providing it with the right type of arguments, initialization etc., while the second can become very clumsy if the decision mechanism requires nesting of such groups. The 'DO' statement in PL/I can handle such situations elegantly as the following example illustrates

FORTRAN

```
      IF(A.EQ.B) GOTO 10
      B=B-1
      A=A+1
      PRINT 5
5     FORMAT (* A AND B ARE NOT EQUAL *)
      GOTO 25
10    READ 15,C
15    FORMAT (Ø12)
      D=A*SQRT(C)
      PRINT 20,A,B,D
20    FORMAT (3Ø12)
25    CONTINUE
```

PL/I

```
IF A=B THEN DO;
            GET LIST(C); D=A*SQRT(C);PUT DATA (B,C,D);
            END;
      ELSE DO;
            B=B-1;A=A+1; PUT LIST('A AND B NOT EQUAL');
            END;
```

Notice that PL/I program does not contain any GOTO's, while the FORTRAN program has 2.

The logical extension of the DO groups is in their use in repetitive calculation. In this respect also, the PL/I DO statement turns out to be more powerful than its counter part in FORTRAN. For example, consider

```
            FORTRAN                          PL/I

      DO 10 I=1,200                  DO:DO P=1.TO 100 BY 0.5
      IF(A.GE.B)GOTO 20              WHILE A < B;
      P=I                                .
      P=P/2.0                            .
      .                                  .
      .                                END DO;
      .
  10  CONTINUE
  20  .....
```

Notice that a single statement in PL/I is equal to as many as 4 in FORTRAN in the above example.

2.1.3 Compound Statements:

Of the two compound statements, namely the IF and ON statement we shall discuss the more important IF statement. The first example of the preceding section already contains a sample use of the IF statement. IF clauses can also be nested as shown in the example below:

```
      IF A>B THEN IF C<D THEN IF T THEN C=0;
      ELSE D=0; ELSE IF B<F THEN K=0; ELSE B=0;
      ELSE IF P+Q<M THEN M=P+Q; ELSE T=0;
```

The equivalent FORTRAN program would be as follows:

```
      IF(A.LE.B)GOTO 1
      IF(C.GE.B)GOTO 2
      IF(T)GOTO 3
      D=0 GOTO 4
      GOTO 4
  3   C=0
      GOTO 4
  2   IF(B.GE.F) GOTO 5
      K=0
      GOTO 4
  5   B=0
      GOTO 4
  1   IF((P+Q).GE.M) GOTO 6
      M=P+Q
      GOTO 4
  6   T=0
  4   CONTINUE
```

2.1.4 Procedures and Blocks: In the hierarchy of PL/I program structure, procedures and blocks occupy the top positions. In programming complex problems, it often becomes necessary to execute a set of statements repeatedly with different sets of input data. Also it becomes convenient to delimit the scope of applicability or uniqueness of a name so that names may be non-unique within a program and yet be well defined locally. These two requirements have been adequately met in PL/I with the introduction of the block and procedure features. We shall first discuss the concept of the Procedure and then that of the block.

Each procedure is surrounded by its own header and trailer statements serving to identify the body of the procedure to the compiler. The label(s) associated with the procedure header is/are the one(s) by which it is referenced and is/are termed the primary entry point(s). A procedure may also have several secondary entry points from where execution may be continued. Procedures are never executed in sequence i.e. they must be invoked by a specific occurrence of the name of the procedure. All names encountered within the body of the procedure are not 'known' outside the procedure except when they are given the attribute EXTERNAL. Procedures may be stacked one after another, each forming a separate entity by itself or they may be nested. In the former case, they are known as external procedures while in the latter case they are known as internal procedures. External procedures are similar to FORTRAN subroutines in the sense that they can be compiled separately and used with different programs. But the similarity cannot be carried much further be-because, as we shall see presently, the procedure concept is a much more powerful concept than the subroutine feature. External procedures may be invoked from any point in a program whereas internal procedures can be invoked only if the immediate predecessor is active. Further, the procedure may be classified as a subroutine procedure or a function procedure depending on whether the referencing is by a call statement or by its occurrence as an operand in an expression.

The 'block', also called a 'begin block', is mainly used for delimiting the scope of names in a program. They are structurally similar to procedures but cannot be invoked from out-of-line and can be entered strictly in sequence or by a GOTO statement whose destination is the label prefixed to the block.

Like the procedure the block has its own header and trailer statements.

The block and procedure form a powerful set of tools in the hands of the PL/I programmer. But one has to exercise caution in the manner in which the block or procedure is invoked in order to ensure that proper transfer of information takes place at the time control is transferred from one procedure to another.

Both the parameter list and the argument list should normally be accompanied by what are known as specifications which assign attributes to the arguments and parameters. If specifications are not provided, PL/I will assign default attributes, but in many cases the default mechanism may be inadequate and may lead to error conditions. For example, consider:

```
P:PROCEDURE (A,B,C);
        .
        .
        .
      END P;
```

Since nothing has been specified about A,B,C they will be assigned default attributes in accordance with the rules of PL/I. When P is called from some other point, the attributes of the argument are, in general, not checked, unless explicitly told to so, to determine if they match those of the corresponding parameters of P. If checking is done, and if there is a mismatch, then conversions will be attempted which may not be always successful. It is safe, therefore, to ensure that either the attributes of the arguments and parameters match or if conversion is to be allowed, explicit directives are given to the compiler so that conversion takes place along predictable lines. When arithmetic expressions are used as arguments, it is likely that the attributes of the result do not match those of the parameters of the called procedure. In order to avoid unpredictable conversions, PL/I allows the programmer to issue directives to the compiler in what is known as an ENTRY list which specifies the attributes of the argument expected by the called procedure. For example, consider,

```
Q:PROCEDURE(M,N,L);
  DECLARE M CHARACTER,N FIXED, L BIT;
  DECLARE P ENTRY (FIXED (5,10),FLOAT (8,4),
                   CHARACTER (5));
      .
      .
      .
  CALL P(M,N,L);
      .
      .
      .
  CALL P((M*N+L), M/N, (L+M+N);
      .
      .
      .
  END Q;
```

In this case the entry list gives a complete idea of what P expects and therefore at the time P is called, the conversion will be performed, if necessary, to the respective targets. This avoids unpredictable results later.

Procedures may be referenced recursively by assigning the RECURSIVE attribute to the procedure name. The following procedure which can be used for calculating the factorial of an integer,is an example of a recursive attribute.

```
FACT: PROCEDURE(N) RECURSIVE;
      IF N=1 THEN RETURN (1);
      RETURN (FACT (N-1)*N);
      END FACT;
```

Let us wind up our discussion on procedures by describing by what is definitely one of the most powerful features of the language, namely the GENERIC attribute. Sometimes it is necessary to have a set of procedures each of which is to be invoked for each of the several types of arguments. Trigonometric functions operate in this manner. Also the procedure to be invoked may depends on the number of arguments. There can be a function procedure which evaluates the roots of a polynomial. The coefficients of the polynomial are to be supplied as parameters. Now for polynomials of 1st, 2nd and 3rd order, simple formulae are available. Thus, for example,

```
DECLARE SIN GENERIC (SIN1 WHEN(REAL), SIN2 WHEN (COMPLEX));
DECLARE ROOTS GENERIC (ROOT1 WHEN (*,*), ROOT2 WHEN(*,*,*),
                       ROOT3 WHEN (*,*,*,*));
```

The WHEN clause determines which alternative is to be selected when the generic family name is referenced.

2.2 Data Structures:

2.2.1 Data Organisation:

Any information that is manipulated by a program may be termed 'DATA'. Data is organisationally divided in PL/I into scalar data items and aggregates of data items.

Scalar data may be either constants or variables as in any high level language. As far as arithmetic constants are concerned, there is not much difference basically between PL/I and FORTRAN except that in PL/I one can direct the compiler to store the constant in a particular fashion (i.e. DECIMAL or BINARY format, FIXED or FLOATING representation etc.). A generalized example would be -10.2367 F-03Y(8,2)I, which means that the Fortran constant -10.2367×10^{-3} must be scaled by a factor of 10^2, represented to 8 positions of precision in binary form and is to be treated as an imaginary constant.

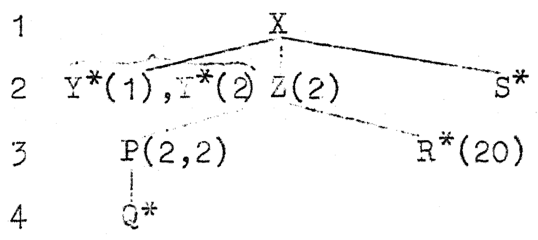
Scalar variable items are also stored in accordance with the scale, mode and precision attributes assigned to them.

Data aggregates may either be arrays or structures or as extended array of structures or structure of arrays. While Fortran users are familiar with the notion of an array, the structure concept is unique to PL/I and is not available in Fortran or Algol. Although some structuring is allowed in COBOL, it is relatively primitive.

A major difference between FORTRAN arrays & PL/I arrays is the manner in which they are stored. In PL/I arrays are stored in row major order. Although this difference is unlikely to affect the normal PL/I user, a systems programmer should take note of this difference while manipulating arrays.

A structure in PL/I is a collection of data items among which a certain hierarchy is maintained. The data items will in general belong to different data types. It is evident that structures may be 'contained' in structures. In other words a structure may be an element of another structure. In such cases, the parent structure is called the major structure while all other structures are called minor structures. An element of a structure may be an array whose elements themselves may be structures. A generalized example of a structure and the equivalent tree is given below:

```
DECLARE 1 X,
            2 Y(2),
            2 Z(2),
                   3 P(2,2),
                            4 Q FLOAT,
                   3 R(20),
            2 S CHARACTER(5);
```



*   Base Elements.

Note that leaves or base elements as they are called, can have attributes appended to them. Base elements should normally use qualifiers in order to make unambiguous references. Cross

<u>Cross Section of Arrays</u>:

The concept of cross-section of arrays is a logical extension of the subscripting notation. Its meaning and use are best illustrated by an example,

```
A(I,J,K,*)  =  A(J,*,J,K) + A(*,I,J,K);
```

In the above statement, the left hand side is said to be a one dimensional cross-section of the 4 dimensioned array A. Its elements range from A(I,J,K,N) to A(I,J,K,M) where N and M are the current lower and upper bounds of the 4th subscript. The dimensionality of the cross-section is determined by the number of asterisks appearing in the subscript list.

While we are dwelling on the subject of arrays and structures, it is worth while to discuss three features of PL/I which facilitate a high degree of manipulation with data aggregates. These are the DEFINED, LIKE and BY NAME attributes.

<u>DEFINED Attribute</u>:

Although the DEFINED attribute may be used with scalar data items also, it is more useful in the case of data aggregates. Quite aften, programming situations demand that we be able to refer to a piece of data by more than one name. This is done by declaring the variable in question in the usual manner and then following it up with a 'Defined' attribute and the name of the variable with which our original variable is to be synonymous.

Consider the statement DECLARE A(10,10), B(10,10) DEFINED A; this will cause the compiler to allocate space for A but for B it will not allocate any space. Each reference to B will be treated as a reference to A. Thus A is the defining variable while B is the defined variable. It is not necessary for B to have the same dimensions of A. For example one can write DECLARE A(4,5), B(3) DEFINED A(1,1); which were result in the following equivalences,

[(A(1,1),B(1)), (A(1,2),B(2)), (A(1,3),B(3))]

There are a few restrictions on the declaration of B and A.~~for~~ For instance B should have the same attributes as A, A itself cannot be a cross-section of some array, and A itself cannot be defined in terms of some other array.

In cases where the dimensions of the two arrays involved in a DEFINED attribute, do not match, care should be exercised in avoiding extension of the defined array beyond the bounds of the defining array.

Even more complicated relationships may be set-up between the defined and defining array with the use of 'iSUB' feature. This is best illustrated by an example,

DECLARE A(20), B(10) DEFINED A(2*1SUB-1);

We calculate the relationship between A and B as follows:

B(1) ≡ A(2*1-1) ≡ A(1), B(2) ≡ A(2*2-1) ≡ A(3),

B(3) ≡ A(2*3-1) ≡ A(5), B(4) ≡ A(2*4-1) ≡ A(7)

and so on.

The expression inside the paranthesis specifying the relationship can be any scalar expression in 'iSUB' where 'iSUB' denotes the value of the i-th subscript in the defined array, i taking values from 1 to n when n is the number of dimensions of the defined array. One can easily visualize how this feature can be put to use in several programming situations. For instance, one can selectively operate upon a matrix to obtain certain results, or one can use the 'iSUB' expression as a sort of code which when used to select elements in a matrix will reveal some intelligible information. A person who does not know the code cannot extract the information although he may have access to the matrix as a whole. This is particularly useful in a time sharing environment where many customers may have access to certain files.

Just as 'defined' attribute could be applied to arrays, so can they be used with structures. Entire structure or part of structures may be given alternative names by the use of the DEFINED attribute.

Consider the example,

```
DECLARE 1 A, 2 B CHAR(4), 2 C CHAR (5), 2 D CHAR (8);
DECLARE E CHAR (17) DEFINED A;
```

When reference is made to E, the 17 characters allotted to the structure A will be involved. Here we see that a byte oriented machine has been implicitly assumed although the design guide lines call for complete machine independence. Thus features such as these would be difficult to implement in a word oriented machine such as the IBM 7044.

LIKE Attribute:

Frequently, we find that two structures have the same hierarchy and the elements have the same attributes. In such cases it would be tedious to declare the two structures separately and in full especially when the two structures have many minor structures and base elements. With the LIKE attribute it is possible to use a short hand notation for all but the first structure so declared. A very good example * would be,

```
DECLARE 1 SOAP, 2 NAME, 3 TRADE CHAR (5),
                        3 TECH CHAR (8),
                2 DATE, 3 MONTHS FIXED (2),
                        3 DAY FIXED (2),
                        3 YEAR, 4 DECADE FIXED (1),
                2 INVENTORY, 3 LARGE, 4 BOXWT FIXED (5),
                                      4 BOXES FIXED (6),
                             3 GIANT, 4 BOXWT FIXED (5),
                                      4 BOXES FIXED (6),
                             3 KING,  4 BOXWT FIXED (2),
                                      4 BOXES FIXED (6);
```

Now if we had another product, say CAKE_MIX which had similar structure, we could simply write,

```
DECLARE 1 CAKE_MIX LIKE SOAP;
```

Even if only a part of a structure matches with the structure in hand, we could still use the LIKE attribute.

Just as in the case of the DEFINED attribute the LIKE attribute cannot be carried over more than one level. Thus in the above example SOAP itself cannot be 'likened' to some another structure.

<u>BY NAME Feature</u>:

Manipulation with structures can be made selective with the help of the BY NAME feature. In this case manipulation on the set of base elements will take place only if at every level from the base upto the root (excluding the root name) the names of the nodes match. As an example consider,

```
        DECLARE 1 A, 2 A1, 2 B1, 3 D1, 2 C1, 3 K1,
                1 B, 2 C1, 3 K1, 3 B1, 2 A1, 2 B1,
                1 C, 2 A1, 2 C1, 3 K1, 3 B1, 4 B1;
```

```
Now,    A  =  B + C BY NAME; would result in

        A.A1 = B.A1 + C.A1;  and

        A.C1.K1 = B.C1.K1 + C.C1.K1;
```

2.2.2 <u>Data Types</u>:

We have so far discussed data on the basis of their organisation in the computer and a few related attributes. We shall now turn our attention to data classified according how a PL/I program operates upon them. Under this classification we can talk of Problem data types and Program control **data** types.

Problem data consists of Arithmetic and string data types. Every data item which is of the arithmetic type is is assigned a base (or radix of 2 or 10), mode (real or complex) and a scale (fixed point or floating point) and a precision which is machine dependent. As has been stressed earlier, these attributes, can either be explicitly specified by a declaration or they will be assigned default interpretations. We can also specify a PICTURE attribute in which a numerical interpretation will be given to data stored in character or bit string format as opposed to coded format of non Picture arithmetic Data described earlier. A data item which has the PICTURE attribute is complete in itself and does not require any other attributes of mode,scale etc. to be assigned to it. The motivation behind the provision of PICTURE attribute is that commercial computation requires not only considerable editing, but also of late,substantial amount of numerical calculations. This is one of the facts overlooked by designers of other higher level languages. Algol and Fortran, which have powerful arithmetic facilities have poor editing characteristics while COBOL has good editing facilities but rigid arithmetic. It is appropriate to mention that whereas in PL/I attributes are listed for each variable or for groups of variables, in Fortran and Algol, declarations are made according to data types.

FORTRAN

```
INTEGER   A,B,C
DIMENSION A(10),B(10,10)
REAL I
COMMON /COM/I
```

PL/I

```
DECLARE (A(10), B(10,10),C) FIXED (8,0),
        I FLOAT (8,5) EXTERNAL;
```

String Data may be classified as character strings or bit strings. String data are characterised by their length which may be fixed or varying during a program. While character strings usually find their use in editing results of arithmetic computation, bit strings play an important role in performing logical decisions in a PL/I programs. Consider, for example the ALGOL program

```
A: = IF C THEN B ELSE D;
```

In PL/I the same program may be written as,

```
A  = B*C + D* (NOT C);
```

Here C is a bit string of length 1. Another important use of bit string in PL/I is for systems programmers. Languages like Fortran and Algol have no bit manipulation capabilities per se. There are many built in functions in PL/I for string data which allow considerable manipulation and editing operations without having to resort to unnatural methods as in FORTRAN.

Let us briefly discuss the other major data type, namely the Program Control Data Types. Any data that can be classified as label, entry, task, event, pointer, offset or area type can be regarded as Program Control Data. We have already discussed entry data type under program structure. POINTER, OFFSET and

AREA variables are discussed under 'List Processing facilities' while TASK and EVENT are discussed under 'Asynchronous Operations' in this Chapter.

Label data items may be constants or variables. Label constants are used to identify statements (similar to statement numbers in FORTRAN) while label variables which could be arrays or scalars are used to perform program controlled transfers. A label variable has, at any stage in the program, as its value a label constant which may then be used as the destination of a GOTO statement. These label variable can be used to perform a function similar to computed GOTO in Fortran, although they (label variables) are much more versatile than just computed GOTOs. For example,

```
   MAIN:  PROCEDURE OPTIONS (MAIN);
          DECLARE (L, ML)(3) LABEL, (M,N) INITIAL (1);
      L:  I=(M+3*N)/3
          L(I)=ML(N); GOTO L(I);
          .
          .
          .
  ML(1):  M=0; N=1;
          .
          .
          .
          GOTO L;
  ML(2):  M=-3; N=3;
          .
          .
          .
          GOTO L;
  ML(3):  M=3; N=2;
          .
          .
          .
          GOTO L;
          END MAIN;
```

2.3 Conversions:

Consistent with the philosophy of 'Anything Goes', PL/I performs a wide range of conversions in assignment statements, I/O statements, procedure calls etc. It is impossible to cover all the situations where conversions take place, and hence we shall limit our discussion to certain basic principles underlying conversions.

In arithmetic expressions, operations are always performed on coded arithmetic data only. If any of the operands is not so coded, it is first converted to coded form before proceeding with the evaluation of the expression.

Effect of Conversion on Precision:

If the precision of the operands in an infix operation differs, no conversion will be done. In case of floating point operations the result will have the precision of the greater of the precision of the two operands. In case of fixed point operations, precision of the result depends on the type of operations and the precisions of the two operands. In every case any necessary truncation will always be made towards zero.

Mode Conversion:

Complex items when converted to real mode, lose their imaginary part and only the real part is retained. When real items are converted to complex items, the result has a zero as the imaginary part. In mixed mode operation any real item will be converted to complex item.

Scale and Base Conversion:

The conversions here are fairly involved and no generalization can be made. For particular cases, the specification manual* should be consulted. Conversions may also take place between arithmetic and string types of data or between character and bit string types.

When bit strings are converted to character type then each bit is converted to the corresponding character and the result has a length equal to the original bit string length.

When character strings are converted to bit string the character string must contain only 0's and 1's, otherwise ~~x~~a conversion error will be raised.

When character strings are converted to arithmetic type, they may contain only those characters which can be coded into arithmetic form.

When bit strings are converted into arithmetic form they are interpreted as though they were unsigned binary integers.

Arithmetic to character strings have no special properties and follow the rules of list directed I/O. When arithmetic data is to be converted to bit string, they are first converted to a real binary fixed point number. The bit string thus resulting, is left justified and has a length equal to the precision of the binary number into which the arithmetic item was first converted.

*PL/I Language Specifications (IBM) Order Number GY33-6003-2

2.4 Storage Classes in PL/I:

Four classes of storage are possible in PL/I.

The AUTOMATIC attribute serves to denote the class of storage where the various data items are assigned storage only when the procedure in which they are declared is invoked. This type of storage is dependent on the logic of the program and is the normal or default class assigned to variables whose scope is INTERNAL or whose scope is unspecified.

The 'STATIC' attribute when assigned to a variable causes allocation of storage at compile time itself and such a data item occupies fixed locations in memory at all times during the execution of a program. This is similar to the storage allocation of Fortran variables and Algol's OWN variables. If a data item is EXTERNAL in scope then it is assumed to have the 'STATIC' storage class if no other storage class is specified.

The third type of storage class allows a programmer to allocate and free storage space according to his choice. Repeated allocations of such data items cause previous 'generations' to be pushed down so that the value of the variable available for manipulation is the one corresponding to the last invocation. Similarly the storage for a variable may be released or 'freed' repeatedly to access the older generations of the same variables. This class is known as the CONTROLLED class of storage and represents a significant departure from the conventional methods, of storage allocation of other high level languages.

The fourth class of storage is known as the BASED class. Every based variable has a pointer associated with it called the pointer variable. By setting the pointer to different addresses in memory the value of the BASED variable can be set to the values of different variables. In other words, no memory storage need be allocated to a BASED variables, for it takes on the value of the variable currently pointed to by its pointer. It is mainly used in list processing and is described in Section 3.

2.5 Data Transmission:

PL/I provides a versatile repertoire of I/O instructions which to date have not been implemented in full in any of the machines built so far. This in itself speaks for the level of sophistication reached by the language as regards data transmission to and from the processor.

The I/O in PL/I is designed to interact closely with the supervisory software under which it functions and to keep track of input sources and output destinations.

Basically, there are two types of data transmission in PL/I - stream oriented and record oriented.

Stream Oriented I/O:

In this mode of transmission, data is considered to be available in a continuous stream of information with no conceptual breaks. Thus, when a command is issued to 'GET' a piece of data, then that piece of data is located from the input stream at a point where the last command to input was terminated. Similarly when a command is issued to 'PUT' a piece of data on an output medium, then it will be written from where the last PUT command ceased to write. Notice the difference between Fortran READ/ WRITE statements and PL/I's GET/PUT command. In Fortran every READ/WRITE command begins a new record, effectively terminating the old record when the last READ/WRITE command ceased to read/ write.

The second mode of transmission is known as the record mode of operation. In this mode, it is assumed that data is prearranged in chunks of information which is then transmitted to or from the processor. The transmission does not involve any prior conversion of data, but is literal, whereas in case of stream oriented I/O conversion will be performed.

It is evident that record oriented I/O will be faster since a) no conversion is performed and b) it explicitly recognises breaks in the incoming or outgoing data. Thus it is an ideal mode of operation when we are dealing with intermediate results which have to be stored temporarily on secondary storage medium. The stream oriented I/O has also its advantages as it allows I/O to be performed with or without editing and permits the novice to specify the actual names of the variables along with their values in the I/O data stream.

The File Concept:

Although the file concept is familiar to most ~~fa~~Fortran users, it is not quite the same in PL/I. A file in PL/I has a symbolic name just as any other variable, (as opposed to Fortran files which are referred to by logical units) and can be broadly classified as Input, output or Update files. An input file may not be used as an output file and vice versa. An update file can be used both as an input and output file.

We shall conclude our remarks on PL/I I/O by stating PL/I file declarations do not carry any specification regarding symbolic or physical storage unit on which the file is to reside. They do not even specify the type of storage medium to be used as repository of the file. All this must be kept track of by the supervisory system. Thus in so far as I/O facilities are concerned a high degree of machine and configuration independence has been incorporated. But this has its own problems in implementation as we shall see later.

2.6 Asynchronous Operations in PL/I:

We shall now briefly touch upon the features in PL/I which provide for execution of a program as a set of asynchronous tasks. Provision is made for the following type of operations:

1. Creating and terminating a task.
2. Synchronising a task.

3. Testing whether a task is complete or not.

4. Assigning and changing priorities of a task.

The definitions of the term synchronous and Asynchronous operations are given in Appendix R . The reader is urged to clarify the meanings of these terms before reading further.

Creating a Task:

It is necessary for at least one task to exist when a PL/I program is started. This is called the major task. Tasks which are initiated by the major task are known as minor tasks. A task is initiated by specifying a TASK or EVENT and/or PRIORITY option with a call statement. The procedure, thus called, will be executed asynchronously with the calling program.

```
Example:  DECLARE B TASK;
               .
               .
               .
          CALL PROC (A1,A2,A3) TASK (B);
               .
               .
               .
```

The execution of the calling procedure is known as the attaching task and the execution of the invoked program is known as the attached task.

Enquiring the status of a Task:

```
DECLARE E1 EVENT;
CALL PROC (A1,A2,A3) TASK (B) EVENT (E1);
```

When procedure PROC is initiated E1 is set to 0 and when ~~it~~ it is completed, it is set to 1. Thus the attaching a task can enquire about the status of the termination of Task B.

TASK SYNCHRONISATION:

This is achieved by means of the WAIT statement whose use is illustrated below

```
P : PROCEDURE MAINS;
    DECLARE DO TASK, (E1, E2) EVENT;
        .
        .
        .
    CALL A TASK (DO), EVENT (E1);
        .
        .
        .
    WAIT (E1);
        .
        .
        .
    END P;
```

In the above statements, the main task procedure proceeds smoothly after initiating task 'DO' until it encounters the WAIT statement. Here further processing in the major task is suspended till task B is completed whence E1 becomes equal to 1.

Terminating a Task:

A task is terminated when the task encounters an EXIT or STOP or RETURN or END statement.

Priority in Asynchronous Operation: In many on line processing systems, situations may arise which compel a task to be suspended and which require the execution of another task.

Such situations are handled in PL/I by assigning priorities to tasks.

The priority for a task is set by the statement

PRIORITY (task name) = expression;

or CALL P(A, B) TASK (T) PRIORITY (exp);

The expression in either case is evaluated and converted to an integer which is then assigned to the TASK as the priority of the TASK.

2.7 Compile Time Statements:

In PL/I provision has been made to instruct the compiler to modify the source text before actually beginning the translation. Such instructions are known as Compile Time Statements as they are executed at compile time itself.

They are similar in structure to other PL/I statements and are distinguished by percentage symbol (%) before the statement beginning. The main set of instructions in this category are the DECLARE, GOTO, IF, DO and the PROCEDURE statements. These statements may be inserted anywhere in the program. Each of the statement is illustrated by an example:

Declare Statement:

```
%DECLARE A CHAR;
%A = 'ACHANGED' ;
```

This will cause every occurrence of A in the program to be replaced by 'ACHANGED' if this instruction is given before any usage of A in the program. Some times a single replacement may trigger off a chain of such replacements and hence repeated scannings will be made till no further replacements are possible. Variables taking part in Compile time statements must be compile time variables and any compile time expression can contain only integer constants.

<u>Compile Time GOTO</u>:

Sometimes a portion of the text may not contain any compile time executable statements. In that case, or in case the programmer deliberately does not want certain compile time statements to be executed, he may use the GOTO statement.

Example,

```
MAIN :PROCEDURE OPTIONS(MAIN);
         %S1   ;    %S2 ;
         %L   : IF A+B=C THEN %GOTO L1;
                         ELSE %GOTO L2;
              .
              .
              .
         %L1 : S3   ;
              .
              .
              .
         %L2 : S4   ;
              .
              .
              .
               END ;
```

The above example also shows the use of the compiler time IF statement. Depending on the boolean value of (A+B=C) either L1 or L2 is executed.

The Compile Time 'DO' Group: This is usually used to minimise execution time at the cost of some more work during compilation and a little more object code.

Example:

```
%DO I = 1 TO 10 ;            Modified Text
    X(I) = X(I)+1;           X(1) = X(1)+1 ;
%END ;                       X(2) = X(2)+1 ;
                               .
                               .
                               .
                             X(10)= X(10)+1;
```

The DO index must be a compile time variable and the WHILE clause is not permitted.

The Compile Time Procedure:

When the preprocessor which handles compile time instructions, comes across a compile time procedure, it bypasses it entirely and resumes scanning after the end of the procedure. When a reference is made to that procedure in a compile time statement, then it will be executed. Compile Time procedures may be invoked by non compile time statements also.

Example:

```
MAIN: PROCEDURE OPTIONS (MAIN);
     %P:PROCEDURE (A,B,C) CHARACTER;
       RETURN (A '+' B '-' C);
     %END  ;
        .
        .
        .
        Y = D+P(X,Z,10) ;
          .
          .
          .
        END ;
```

The preprocessor will replace the relevant statement by
Y = D+A+B+-10;

3. PL/I and List Processing:

When PL/I was conceived of in late 1964, the originators had not included list processing facilities in the language. List processing facilities were added almost as an after thought sometime in 1965-66. The based class of storage, as mentioned earlier, has great relevance to list processing. In the discussion presented below, the reader is assumed to be familiar with list structures.

In lists, the underlying principle is to break the relationship between physical and logical arrangement of certain items of data. In other words, we do not want to be tied down by having contiguous chunks of storage space alloted to a set of data items. By removing this restriction, lists enable us to shift data around in a convenient manner. Whenever storage is allocated to a based variable by an ALLOCATE statement the allocation of such storage (which is different from previous generations) must be indicated by setting the corresponding pointer variable. For example,

```
          DECLARE  B1, B2 BASED (POINTR);
          ALLOCATE B2 SET (POINTR);
```

The SET key word is used to update the value of the pointer to the current storage address of the variable B2. More rigid control over allocation of based variable may be exercised by specifying an area where allocation is to be made. This is done as follows,

```
        DECLARE A AREA AUTOMATIC (FIXED (10,0),...FLOAT(8,5),
                                  CHAR (30)...);
```

This declaration reserves enough space for A to hold as many items as are specified within parenthesis.

Now if we allocate a variable in this area by

```
        ALLOCATE B2 IN (4) SET(POINTR);
```

this would cause allocation of B2 in A if there is enough space in A. If there is not enough space available, an error condition will be raised.

We shall illustrate the use of the above attributes in generating a unidirectional list by the following example,

```
        /* THIS PROCEDURE IS USED TO APPEND AN ELEMENT POINTED TO
           BY 'EP' TO A LIST WHOSE NAME IS 'ELEMENT'*/
               LIST: PROCEDURE (EP);
               DECLARE EP POINTR, 1 ELEMENT BASED (EP),
                                  2   P   POINTR,
                                  2   V   VALUE;
               P=NULL
               IF TAIL=NULL THEN HEAD,TAIL=EP;
                            ELSE P(TAIL), TAIL=EP;
                    END LIST;
```

In the above example, the first time a call is made to LIST the HEAD and TAIL pointers are set to the first element. Subsequently the TAIL is moved to point to the latest element added while

HEAD is undisturbed. Also the pointer P is correspondingly updated to form a unidirectional list.

The NULL built in function when used for initialization creates an empty cell and returns the address of the cell as its value. When used in the IF statement, it is used to check whether TAIL is pointing to a null list.

Yet another attribute which is useful in list processing is the OFFSET attribute. This is primarily used in conjunction with a based variable in a based area.

For example,

```
DECLARE ROOM AREA BASED (P1),
        RUSH OFFSET (ROOM),
        VAR FIXED (2) BASED (P2);
ALLOCATE VAR IN (ROOM) SET (P2);
```

OFFSET will now contain the amount of offset between the start of the area ROOM and the variable VAR.

OFFSET variables differ from POINTER variables in one important aspect and that is OFFSET variable always gives the relative displacement while POINTER gives the absolute m/c address of a based variable.

In conclusion we can say that in PL/I list processing can be done as elegantly as in any other language while retaining all the data manipulation capabilities of other high level languages.

CHAPTER II

OVERVIEW OF THE PL 7044 COMPILER

This Chapter discusses the choice of a subset of PL/I the overall organisation of the compiler, its limitations and some techniques used in the implementation.

1. The Choice of a Subset:

One of the first decisions that one has to take in a compiler writing project is the extent to which the implemented language is to be loyal to the original language specifications. It should be borne in mind that, more often than not, language specifications are made without considering the implementation hurdles that certain aspects of the language may pose on existing machines. We wish to point out that PL/I is not an exception in this respect, rather, as has been discussed earlier, PL/I is more machine independent than most other high level languages. Thus, one is left with the difficult task of pruning the original language, in order to inject a high degree of viability in the implementation of the language.

Our decisions, in formulating the subset of PL/I, hereinafter to be referred to as PL 7044, have been guided by the principle of maximising the number of user facilities in the subset. We set out by aiming for a full PL/I compiler. As we met hurdles, we tried to overcome them, and if we failed for one reason or other, we dropped that feature which caused the hurdle.

Thus our method of attack has been 'to cross the bridge when we come to it'. We must admit that this method may have cost us in terms of a slightly inefficient code, and a rather unwieldy organisation. But the reader will readily accept this line of approach if we set forth the constraints under which the implementation was carried out.

Firstly, the time available to us was limited. It would have been difficult for us to complete the formulation before we started the coding, It was necessary for us to carry on formulation and coding side by side so that we could be sure that the ideas we put down on paper could indeed be programmable on the IBM 7044 without exceeding the limitations of the computer like memory space, execution time etc. Secondly the lack of previous experience in a project of this type, slowed down the progress to some extent. It was difficult to foresee problems before we were actually confronted by them. Lastly, the lack of detailed literature on implementation of the language on various machines forced us to constantly change our strategy on an ad hoc basis. This is one of the reasons why global optimisation and index register optimisation were not attempted.

2. Organisation of the PL 7044:

2.1 Choice of Number of Passes:

Compilers are normally organised in a number of distinct passes or phases, each phase carrying out a well defined task. The choice of the passes is, then, crucial in determining the overall efficiency of code generated and the speed of the compiler itself. These two factors are contradictory in nature and usually a compromise is struck depending on the objectives of the designers. Those in favour of lesser number of passes, often advance the argument that by 'doing as much as possible in a single pass' one reduces the book-keeping involved and avoids the problem of evolving suitable intermediate codes between passes. Also the fewer the number of passes, the lesser the quantum of I/O activity, in general, and to that extent the compilation is faster. Following this line of thought, a two pass compiler was contemplated for PL 7044. But very soon it was found to present insurmountable hurdles as outlined in the next section of PASS 1. It is worth while to mention here, some of the advantages that accrue from a multipass compiler.A rule of thumb suggests that the greater the number of passes the more efficient is the code likely to be. Also, sometimes larger number of passes may be the only feasible answer to implementation of certain high level languages like PL/I, COBOL etc. on machines having small fast access memory and a large not-very-slow secondary memory. For example, if one were to try and implement PL/I on an IBM 1620 m/c it would be fair to guess the number of passes at around 20 or more ! On the IBM 7044 it was finally decided to divide the compilation work into 3 passes, the descriptions and necessities of which follow.

2.2 Pass I:

The primary aim of the first pass is to resolve certain apparently ambivalent situations in the program text. To make the point clear let us consider the following examples,

```
Example 1     PROGRAM A              PROGRAM B
                  .                      .
                  .                     . .
                  .                      .
      . . ..  2 + 3.I  . .     . .  2+3.  +I    . .
                  .                      .
                  .                      .
                  .                      .
```

In PROGRAM A '2+3.I' denotes a complex constant, whereas in PROGRAM B we have a simple addition of a 2 real constants and a variable I. If we wish to resolve the uncertainty at lexical level, it would be necessary for the lexicon to look-ahead as many as 3 lexical units ahead every time it encounters a simple constant. This will slow down the otherwise fast lexicon.

Example 2:

```
DECLARE (REAL, INTEGER, COMPLEX) STATIC;
DECLARE (REAL, INTEGER, COMPLEX) = 0;
```

As PL 7044 does not have any reserved symbols but only certain key words which could be used as ordinary identifiers also, in the above example it is impossible to say anything about the nature of the statement till we encounter the '=' sign or the keyword STATIC. It would be illogical to argue that the symbol table will come to our rescue in such situations.

Even if the symbol 'DECLARE' were declared as a 3 dimensioned variable and is available as such in the symbol table, we cannot categorize the statements as an assignment or declaration statement merely by scanning the first symbol 'DECLARE'

Example 3:

```
           PROG - 1                          PROG-2
MAIN:PROCEDURE OPTIONS (MAIN);  MAIN:PROCEDURE OPTIONS (MAIN);
        .                               .
        .                               .
        .                               .
     DECLARE B REAL ;                DECLARE B REAL ;
        .                               .
        .                               .
        .                               .
     CALL SIN (A,B);                 CALL SIN (A,B) ;
   A:D=0 ;                           END MAIN ;
     END MAIN ;                   SIN:PROCEDURE ;
 SIN:PROCEDURE ;                        .
        .                               .
        .                               .
        .
```

In both the above programs, upto the statement 'CALL SIN(A,B) there is no information about the symbol A, therefore no conclusions can be drawn as to its nature. In program-1 it turns out to be a label constant, while program-2 expects default attributes to be assigned to A. The problem becomes more acute because of the conversions that have to be performed when arguments and parameters in procedures do not having matching attributes. The problem becomes doubly compounded when we add a DECLARE statement in the main procedures of PROG-1 and PROG-2 as follows:

```
          DECLARE SIN GENERIC (SIN1 WHEN (label, real),
                               SIN2 WHEN (real, real));
```

The first program would expect a call to SIN1 be made, while the second would require a call to SIN2 be made.

Thus, in all these examples, the first pass can come to our rescue. In the first example it would have collected enough information about the nature of the constant, so that when second pass requires a lexical unit at that point in the program, there would be no difficulty in supplying it with a complex constant in one case and a real constant in the other. In the second example, the classification of statements carried out by the first pass helps in resolving the ambiguity for the 2nd pass. In third example the first pass would have collected all the labels of each block and made it available in a group to the 2nd pass with the result that at the beginning of each block the 2nd pass knows exactly which symbols constitute label constants and which of them have to be assigned default attributes. Certain other duties are also assigned to the 1st pass to make things easy in the 2nd pass. These are detailed in the Chapter on 'FIRST PASS PROCESSOR'. We have enough evidence in hand to justify the existence of this pass.

2.3 Pass - II:

In this pass, most of the syntax analysis and all of the semantic interpretation are carried out. But for the limiting factor of the memory space available, code generation could easily have been included in this pass, rendering the 3rd pass redundant.

The syntactic analysis is a decentralized process and is carried out by several statement decoding routines. This strategy has a clear advantage of being fast compared to a centralized syntactic analysing scheme. The advantage accrues from the fact that each decoding routine not only checks the syntax but also assigns the semantic interpretation simultaneously. Such a possibility is precluded in the case of the centralized syntactic analysing scheme. A slight disadvantage of decentralization may be the larger core storage requirement, but, in any case, since we need a separate pass for code generation, it is not worth while to save a few location and lose in time.

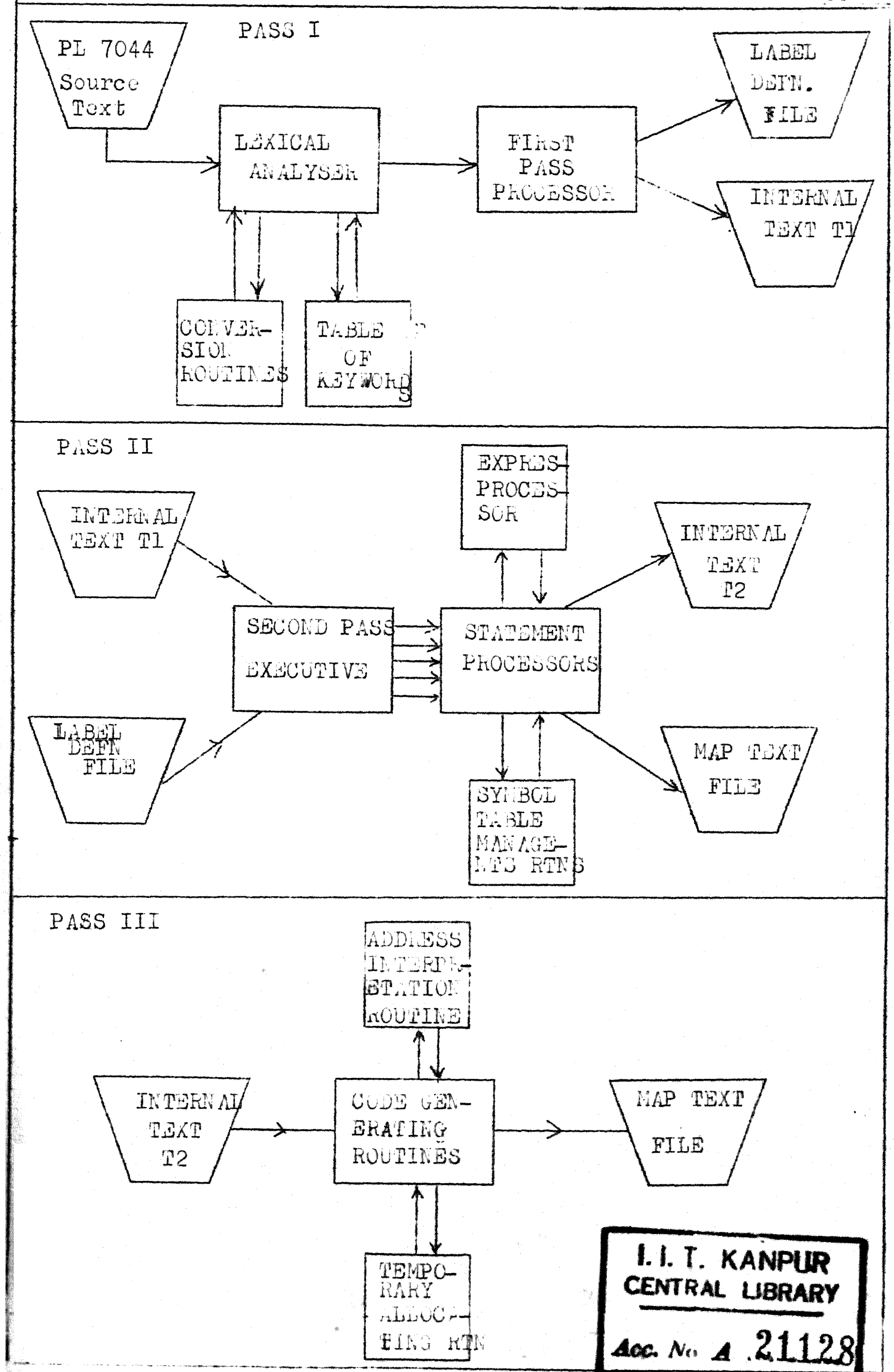
The second pass has a set of symbol table management routines which are used by the decoders for getting/adding information from/to the symbol table. Since in PL/I language scalar expressions could be used at every conceivable point in the program, the expression handler is another facility which is utilized by the decoders. The expression handler has 3 interrelated routines - the buffer handler, arithmetic processor and the sorting routine. The motivation for this type of organization of the expression handler, and the tasks performed by the 3 routines have been detailed in Sec. 3.2 of this Chapter.

It is worthwhile to mention at this point, that even though the bulk of the code is generated by the 3rd pass, a small portion of it is generated in the 2nd pass also. Coding which is independent of the logic of the program falls under this category. Data lists, constants, file declarations etc. form a part of the code outputted during the second pass. This is in keeping with the basic philosophy of doing as much as possible in every pass.

2.4 Pass - III:

As has already been mentioned in earlier sections, the 3rd pass generates the final coding in a form acceptable to the MAP assembler. Apart from code generation, optimisation of temporaries is also carried out in this pass. No other complications are tackled at this stage and hence the 3rd pass constitutes, logically, the simplest pass of the three passes.

Four work units are required by the PL 7044 compiler. The output of the first pass is held on Work Unit 1. This output is read in by the executive of the second pass. The label definitions, block predecessor table and the DO table are outputted on Work Unit 2. In the second pass, the tables are initially read into pre-assigned locations, and at the beginning of each block, the label definitionscollected by the 1st pass are entered into the symbol table. The intermediate code generated by PASS II is outputted on Work Unit 3 while the final MAP coding is made available on Work Unit 4. It is not possible to further optimise


PASS I
PL 7044 Source Text
LEXICAL ANALYSER
FIRST PASS PROCESSOR
LABEL DEFN. FILE
INTERNAL TEXT T1
CONVERSION ROUTINES
TABLE OF KEYWORDS
PASS II
INTERNAL TEXT T1
EXPRESSION PROCESSOR
INTERNAL TEXT T2
SECOND PASS EXECUTIVE
STATEMENT PROCESSORS
LABEL DEFN FILE
MAP TEXT FILE
SYMBOL TABLE MANAGEMENTS RTNS
PASS III
ADDRESS INTERPRETATION ROUTINE
INTERNAL TEXT T2
CODE GENERATING ROUTINES
MAP TEXT FILE
TEMPORARY ALLOCATING RTN

on the number of Work units because during the 2nd pass, all the four work units are active.

The three passes may be edited on the System Library as three phases operating under the Processor Monitor IBJOB. The IBJOB will hand over control to the 1st pass via the System Loader and each of the passes will return control to the System Loader with a request to load the next phase.

3. Implementation Details:

3.1 Memory Management:

3.1.1 Storage Classes:

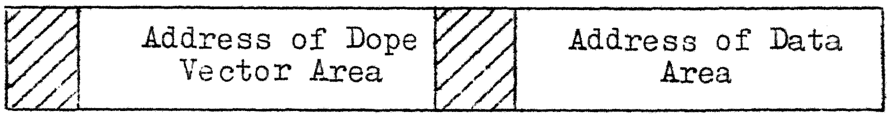
PL 7044 allows a programmer a choice of two storage classes - STATIC and AUTOMATIC. The other two classes of storage namely, CONTROLLED and BASED are not available because they can lead to fragmentation of memory in the absence of hardware facilities like segmentation and paging. The fragmentation will arise in a linear memory space, such as is available on the 7044, in the following manner. The CONTROLLED class of storage allows a programmer to allocate and free memory space for a particular variable in accordance with his requirements. He may, for example, create several generations of the same variable and then release storage for any or several of these generations. Now the order in which he creates successive generations need not be the exact reciprocal of the order in which releases them. In other words, in many programming situations the last-in-first-out principle may not be followed and this can leads to fragmentation of memory space.

PL 7044 compiler analyses storage requirement for the entire program and schedules the allocation of storage as it is needed. At run time, a general pool of storage cells is maintained which receives storage cells as when they are released and allocates them to other data items as and when they are demanded. Thus a data item may not be allotted the same memory space at two different allocations. The difference between the two classes of storage STATIC and AUTOMATIC is intimately connected with the partitioning of memory into blocks and will be discussed in Section 3.1.3.

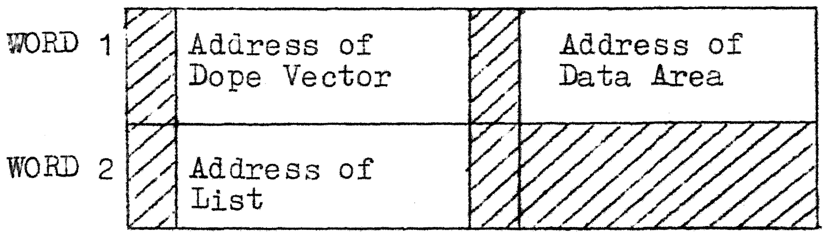
3.1.2 Representation of Various Data Items in Memory:

The rules for forming constants and their internal representation are given in Appendix B and will not be touched upon here, since they are self explanatory. Appendix D lists variable items under three headings - arithmetic variables, string variables and label variables. The internal representation of these types have also been given in detail and will not be discussed here. We shall concentrate our attention here to data aggregates and how they are handled during compilation and run time.

Internal Representation of Arrays: The array is represented in PL 7044 by three chunks of memory space called the header, the dope vector and the actual area containing data. The header area is a single 7044 word whose format is shown below.



This is the format when the array contains complex, floating point, integer, character or bit string data items. In the case of an array of labels with a permitted list (See Appendix D for a description of label variables with a list) the header area consists of two words having the following format



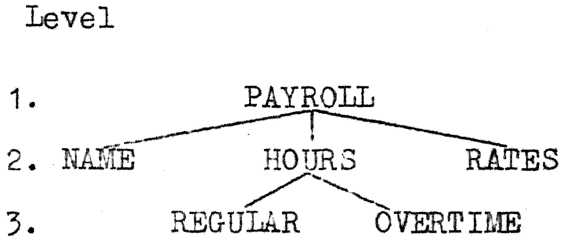
The dope vector area contains information about lower bound, the range and the multiplying factor for each of the N dimensions of the array. It also stores an address which is useful for accessing an element of the array in the data area. Thus the space requirement for the dope vector of an N dimensioned array turns out to be 3*N+1 locations.

<u>Internal Representation of Structures:</u>

The motivation in having structures and the facilities to manipulate them has been outlined in Chapter I. We shall concentrate our attention here to the internal representation of structure on IBM 7044.

A structure is represented in core storage as a linear collection of all base (or leaf) elements if the corresponding tree is scanned from left to right. Thus for example,

```
DECLARE 1 PAYROLL,
          2 NAME,
          2 HOURS,
            3 REGULAR,
            3 OVERTIME,
          2 RATES;
```



will be assigned storage as follows:

NAME	REGULAR	OVERTIME	RATE

Each base element in this collection is treated as if it has been declared as a variable. In fact, at execution time, the structure looses its identity. Each base element is treated as a variable for addressing purposes. Of the two extended data aggregates - arrays of structures and structures of arrays, the former is not provided. Structures of arrays have been taken care of, however, and their internal representation is best illustrated by the following example:

```
DECLARE 1 IITKANPUR, 2 PURESCIENCE, 3 DEPARTMENTS (3),
                     2 APPLIEDSCIENCE, 3 DEPARTMENTS (7);
```

The internal representation of the above structure, which has two arrays as its base elements, is as follows

HEADER OF PURESCIENCE.DEPARTMENTS	HEADER OF APPLIED SCIENCE.DEPARTMENTS

Thus, if a base element of a structure is an array then its header word gets inserted in the linear collection of base elements that we talked of earlier.

Representation of Formal Structure:

A major structure or a minor structure may be passed on as an argument in a procedure call. It is the responsibility of the programmer to make sure that the corresponding parameter of the called procedure is also a structure whose specifications (attributes) and hierarchy match those of the transmitted structure. When such is the case, the formal structure parameter (henceforth to be called formal structure) is once again represented by a linear collection of base elements. Each such base element is then treated as if it has been explicitly declared as a formal parameter. As in the previous case, the formal structure then loses its identity. There is, however, one important difference between normal and formal representation. Whereas in the case of normal structure, no storage cell is allotted to the major structure, in the case of the formal structure, the major structure is allotted one word for purposes detailed below.

When a formal structure (either a major or minor) is transmitted as an argument in a procedure reference, it is necessary to transmit the address of the first base element of the minor or major structure in the parent (or normal) collection of base elements. This facilities uniform treatment of formal structures no matter how many times the same structure or a part of it is transmitted through a number of procedure references. Thus, it is this address of the first base element of the structure (major or minor) in the parent collection that occupies the word allotted to the formal structure parameter. The above discussion is illustrated by the example below:

```
MAIN:PROCEDURE OPTIONS (MAIN);
     DECLARE 1 A, 2 B, 3 C CHARACTER (10), 3 D CHARACTER (5),
             2 E, 3 F (10,10) COMPLEX, 3 G, 4 I (10) LABEL,
             3 J, 4 K (10,10,10), 4 L BIT (10);
        .
        .
        .
     CALL PROC1(A.E);
        .
        .
        .
PROC1:PROCEDURE(M);
      DECLARE 1 M, 2 N(10,10) COMPLEX, 2 X, 3 P(10) LABEL,
              2 Q, 3 R(10,10,10), 3 S BIT (10);
        .
        .
        .
      CALL PROC2  (M.Q);
        .
        .
        .
PROC2:PROCEDURE(X);
      DECLARE 1 X, 2 Y (10,10,10) REAL, 2 X BIT(10);
        .
        .
        .
```

The storage representation of the three structured iis shown in Fig. OC-1

Note that for formal structures M and N, the cells allotted to them point to the first base element under the respective major/minor structure of the parent structure. When A.E is transmitted from the MAIN procedure to PROC1 there is no difficulty in calculating the address of the base element of A.E, namely F. It is equal to (BS.XX+Y)+(3-1) = a_1 say) where BS.XX+Y is the starting address of the normal structure in core. When M.Q. is transmitted

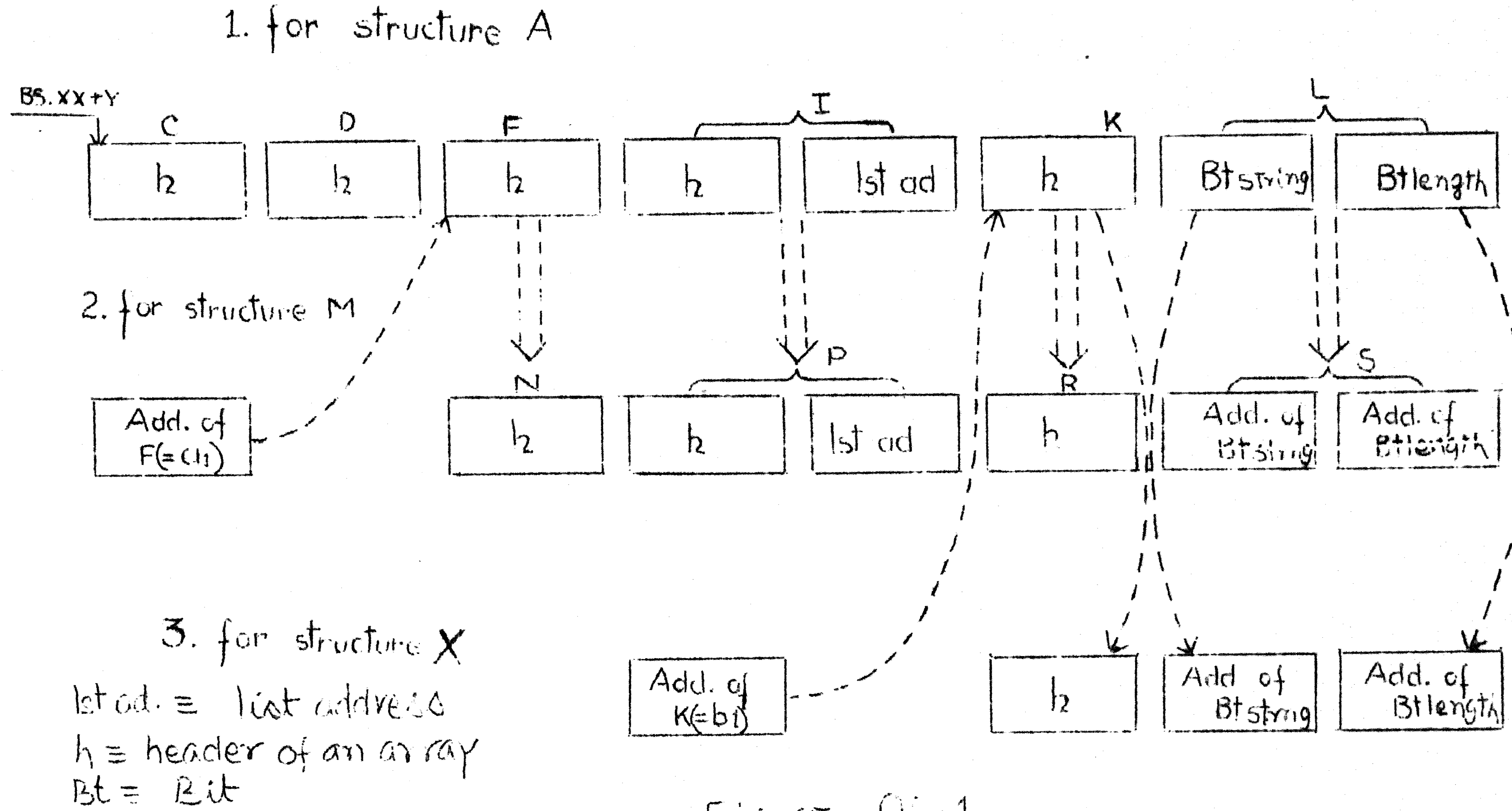

1. for structure A
BS.XX+Y
C
D
F
I
K
L
h
h
h
h
1st ad
h
Bt string
Bt length
2. for structure M
Add. of F(=C1)
N
P
R
S
h
h
1st ad
h
Add. of Bt string
Add. of Bt length
3. for structure X
Add. of K(=b1)
h
Add of Bt string
Add. of Bt length
1st ad. ≡ list address
h ≡ header of an array
Bt ≡ Bit

FIGURE OC-1

to procedure PROC2, the address of element K in the parent collection is arrived at by adding the offset of the first element of M.Q which is R to a_1, i,e b_1 = addr. of K = $a_1+(4-1)$ = (BS.XX+Y) +(3-1)+(4-1) = (BS.XX+Y)+5.

It is, worth while, at this point to comment on the splitting of the header word and the dope vector of an array into two distinct storage areas instead of merging them into one. If we had not resorted to splitting of the two sections, it will be necessary for us to transmit both these areas every time a structure of arrays is passed on as an argument in order to maintain the uniformity of treatment of all format structures. Thus an area equal to 3*N+1 when N is the dimension of the array, will have to be copied every time, resulting in not only wastage of storage space but also in execution time because the copying is done at run time. Thus splitting the two areas avoids wastage of space and time and also allows the above scheme of internal representation of formal structure to function successfully.

3.2.3 Memory Partitioning:

In order to implement the block structure of PL/I, it was found necessary to partition memory software-wise. Before we proceed to detail the partitioning strategy, two definitions seem appropriate.

1. Block Representation: Each block of PL 7044 program is represented and referred to by an unsigned octal integer which is the number assigned to it in lexicographically ascending order. Thus, if the compiler has encountered $(N-1)_8$ blocks before encountering the current block, then the current block will in future be identified by N_8.

2. Predecessor Block: A predecessor block is one that has, between its header statement and ending statement, the entire body of the block whose predecessor we are seeking. An immediate predecessor is a predecessor which is lexicographically closest to the block in question.

PL 7044 allows a maximum of 62 blocks in a program. All variables having the attribute 'STATIC' are considered to be belonging to an imaginary block '0' which is never closed and which encompasses the entire program and thus simulates the effect of a STATIC declaration.

A 'block predecessor table' is created during the 1st pass and is passed on to 2nd pass and finally to the run time x code. The Block Predecessor Table (BPT) carries the block number of the immediate predecessor of each block. Thus the BPT stores the lexicographic descendence relationship between blocks.

Available memory space is divided into 3 main parts,
(1) Scalar storage area (herein after called as FIRST ORDER STORAGE)
(2) Array Storage Area (herein after called SECOND ORDER STORAGE)
(3) Save Storage Area (herein after called THIRD ORDER STORAGE).

The extent of the first category of storage is determined at compile time while the extent of the 2nd and 3rd order storage is determinable only at run time and these categories are allocated storage from the top and bottom of the free core space after the program code and first order storage have been loaded. More about the run time storage management is given in Section 3.2.4.

Scalar Storage Area (First Order):

An address of a scalar variable is an ordered pair, the first member of which is the block number N in which the variable is declared and the second member is an offset with respect to the starting of the storage scalar area of the block N. Scalar storage area may be divided under three heads, namely a) scalar element area, b) array dope vector area and c) temporary area.

Scalar Element Area:

All scalar elements defined in a particular block and the header words of arrays and character string variables are assigned storage in this area.

Array Dope Vector Area:

The representation of an array as described earlier consisted of three different areas. Of these, the dope vector is assigned storage in this area. The starting address of this area is the last location occupied by the scalar element area plus one.

Temporary Area:

The temporary storage requirements of a block constitute this area. The starting address of this area equals the last location of the dope vector area plus one.

Some Notations:

Starting Address of first order area of a block	BS.XX
Starting Address of Scalar Element Area (SEA)	BS.XX
Starting Address of Array Dope Vector Area (ADVA)	AS.XX
Starting Address of Temporary Area (TA)	TS.XX
A location in SEA may be referred to by	BS.XX+<offset>
A location in ADVA may be referred to by	AS.XX+<offset>
A location in TA may be referred to by	TS.XX+<offset>
Last first order storage location of ablock block + 1 ≡	BE.XX

In the above notations 'XX' represents the block number in octal and 'offset' is the position of the location relative to the start of each area.

For each block the following relationships hold,

```
BS.XX  EQU  BE.P(XX)
AS.XX  EQU  BS.XX+#(SEA)
TS.XX  EQU  AS.XX+#(ADVA)
BE.XX  EQU  TS.XX+#(TA)
```

where P(XX) = immediate predecessor block number.

The maximum number of first order storage is calculated as follows:

Step 1:

$$\#(0)=SEA(0)+DPVA(0)+TA(0).$$

Step 2:

At the end of each block the total first order area required till then is calculated as

$$\#(B_i) = \#(P(B_i)) + SEA\ (B_i) + DPVA\ (B_i) + TA\ (B_i)$$

where $P(B_i)$ is the immediate predecessor of the i-th block and SEA, DPVA and TA all have same notation as above,

Step 3:

$$\left.\begin{array}{l}\text{Maximum first order}\\ \text{Storage Required}\end{array}\right\} = \underset{i=0,1,\ldots N}{\text{MAX}} \{(\# B_i)\}$$

Let the maximum of first order storage be represented by MXFSAR.

At the end of the 3rd pass MXFSAR is available to the compiler and it then generates the following code

```
BS.00  EQU  *
       BSS  <MXFSAR>
```

The 3rd pass also prepares a 'block storage table' and outputs it along with other code. This table consists of the starting address and ending address + 1 of the first order area of each block. It is used by the prologue routine during run time to effect saving and restoring of 1st order area in the 3rd order storage area.

The general format of an entry in the block storage area is,

```
PZE  BE.XX,,BS.XX
```

where,

BE.XX and BS.XX have the same connotations as before. Appendix F illustrates all that we have discussed about block structures by working out the actual codes that will be generated for the example in Appendix E.

Saving and Restoring First Order Storage:

As can be seen from the illustration in Appendix F, two or more procedures will share the same storage space if they have the same immediate predecessor. This is because the scalar storage area of one such block is inaccessible to the other, thus enabling the prologue routines to save, use and restore the same physical area over several procedures having the same immediate predecessor. Note that this technique will fail if the 'TASK' attribute were to be implemented because two procedures may be under execution at the same time and this will not permit such overlaps.

Let us now discuss, briefly, how the prologue routines calculate the amount of core storage (from first order area) to be saved in the SAVE area (third order area). At the time of invoking a procedure, the value of (BS.<called-procedure> - BE.<calling-procedure>) is calculated. If it turns out to be positive or zero then no saving need be done, since it implies that the block being opened does not share any first order area with the calling block.

On the other hand, if a negative value is obtained,the prologue routine saves an area equal to $|\text{BS.}\underset{\text{procedure}}{\text{<called>}} - \text{BE.}\underset{\text{procedure}}{\text{<calling>}}|$. This count and the position in the 3rd order area is carried along in the run time stack which is the topic of our discussion in the next section (3.2.4).

At the time of closing a block, and returning to the calling program any area that has been saved during invocation must be restored. This is achieved by consulting the linkage cells of the block in the run time stack (Section 3.2.4). We shall defer our discussion on the exact mechanism of restoring till the end of next section.

3.2.4 Run Time Stack Management:

The entire free core storage available after loading the program code, first order storage requirements, supporting sub-routines from the system library and the I/O buffer requirements is converted into a double ended stack which can be accessed from both ends. Fig. OC-2 shows schematically the runtime stack. The stack comprises the 2nd order storage area of each block described in Section 3.2.3, the 3rd order storage area and certain linkage cells which are vital for transfer of control from one block to another and other book keeping operations which will outlined below.

The general format of the linkage cell is as follows:

▨	L.P.P.	▨	return Addr.
▨	W.A.P.	▨	# of words saved
▨	DOSRNO	▨	Calling Block No.

<u>Notations</u>:

P.P.P. (Present Procedure Pointer) - This is a one word area which contains the address of the first of the 3 locations which constitute the linkage cell of the block which is being currently executed.

L.P.P. (Last Procedure Pointer) - This is the value of the P.P.P. when the invocation of the present block was made.

W.A.P. (Work Area Pointer) - It is the address of the first free location after arrays and character variables (Second order storage Area) have been allocated memory space. This denotes the address of the first free cell available for assigning temporaries requiring more than 2 words.

F.C.P. (Free Cell Pointer) - It is the address of the first free location after multiword temperaries have been assigned.

At the end of each assignment statement or expression using a multiword temporaries, the arithmetic processor generates code to bring the Free Cell pointer back to the value of the Work Area Pointer. This results in a slight inefficiency in the use of available space, since not all the temporaries will be required at the same time and some amount of sharing is possible. But the

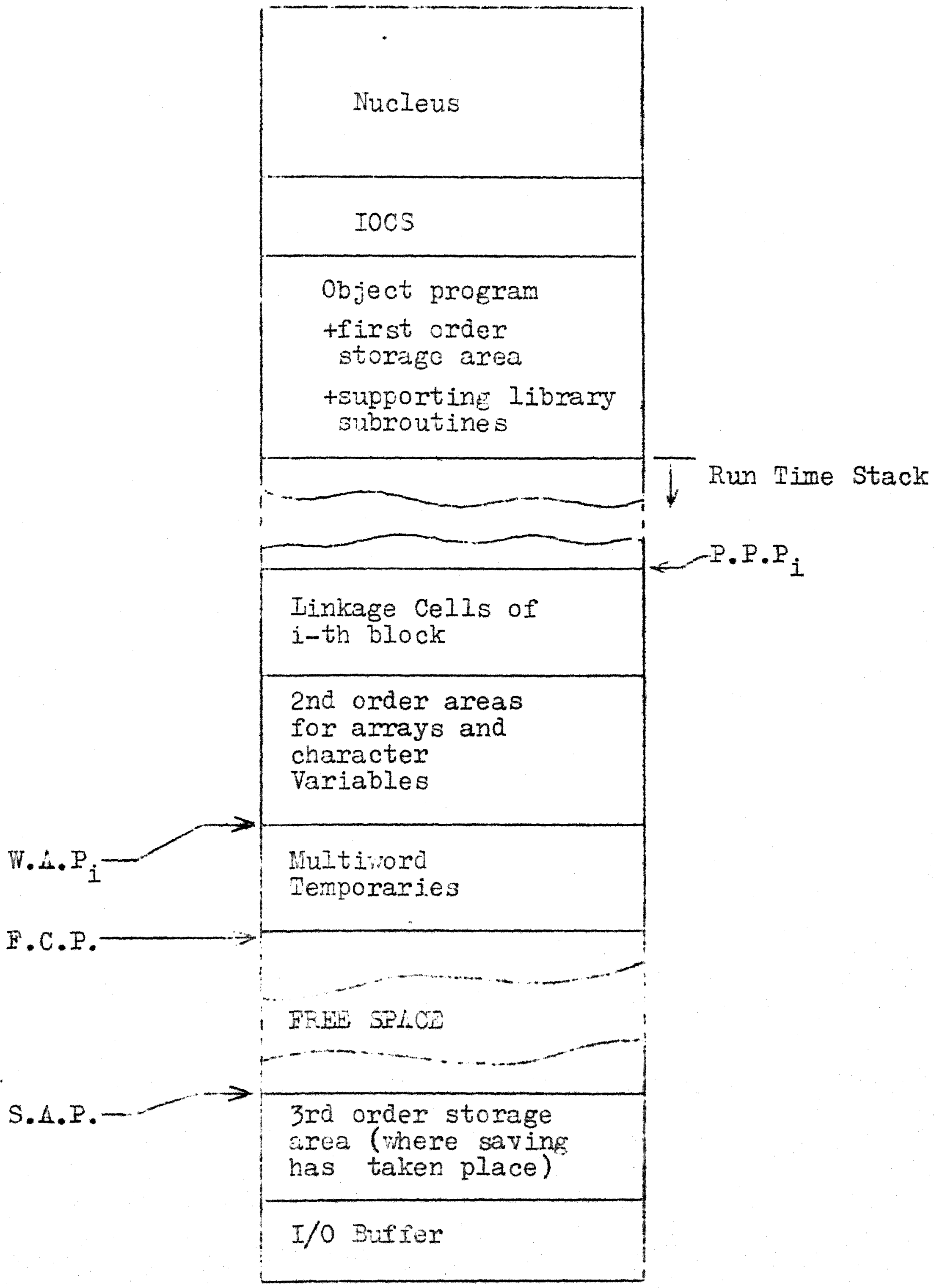

Nucleus
IOCS
Object program
+first order storage area
+supporting library subroutines
Run Time Stack
P.P.P$_i$
Linkage Cells of i-th block
2nd order areas for arrays and character Variables
W.A.P$_i$
Multiword Temporaries
F.C.P.
FREE SPACE
S.A.P.
3rd order storage area (where saving has taken place)
I/O Buffer

Fig. OC-2 : Memory Map at Execution Time Showing Run-Time Stack.

logical simplicity of the present scheme of alloting all the temporaries at one shot has been given more weightage in our considerations of the two methods. Also, if the temporaries are not allocated and released according to the last-in-first out principle, it would be difficult to keep track of the resulting fragmentation of memory space.

S.A.P. (Save Area Pointer) - It is the address of the location upto which the stock is filled from the high core side with 1st order storage saved during procedure invocations.

<u>Mechanism of the Linkage Cell</u>:

1. L.P.P.- At the time of closing a block we must reset the P.P.P. value to the address of the linkage cell of the block which invoked the current block. This is precisely the content of L.P.P.

2. Return Address: For a 'begin' block, this value is zero, since it cannot be called out-of-sequence; for a procedure it contains the address complement of the point of invocation. This is necessary for continuing execution after a called procedure has been deactivated by a RETURN or END statement.

3. W.A.P. - This pointer is to be saved in the linkage cell of the current block, every time a procedure invocation is made. This is because the called procedure itself will alter the value of W.A.P. and in order to retrieve the value of W.A.P. after call is over, we must save the current value of W.A.P. It is evident that while entering a 'begin' block such a precaution is not

necessary since a begin block cannot be invoked from out-of-line.

4. # of words saved: It stores the number of words saved, if any, at the time of invoking the present block. This is needed for determining if any restoration is to be carried out when closing the block.

5. DOSRNO: It is the serial number of a 'DO' group if the invocation has been made from inside a DO group. This is to ensure that the distination of a possible GOTO statement in the current procedure is a valid destination.

6. Calling Block No: With the help of this information and the number of words saved if any, the run time stack management routines carryout restoration of 1st order storage area saved.

3.2 Expression Processor:

As the name suggests, the expression processor is primarily concerned with the generation of object language codes for expressions in PL 7044. The code generation of the assignment statement is also done by the expression processor. The occurrence of an expression is permitted in practically every statement and at every conceivable point in the program. Each situation delimits the expression by a different pair of terminal symbols. Hence the expression processor must recognise the situation from where it is called and output code according to the needs of the situation. Thus, the factors one has to bear in mind while designing an expression processor may be enumerated as under:

(i) Ambiguity regarding the use of an operator: In PL 7044, '=' is used for both assignment and relational comparison. The choice, to be made between the two meanings, is highly context dependent. Similar is the case with delimities like ',' '(' ')' etc. Hence the expression processor should be designed to distinguish between multiusages of operators and delimiters.

(ii) Recognition of the domain of the expression: As mentioned in the opening remarks, the expression processor must recognise different pairs of end markers for different calls. For example, in DO A(I,J) = ... ; the expression processor will be called into action after the 'DO' has been recognised by the 'DO' handling routine. In this case the input for E.P. is delimited by 'DO' on one side and '=' on the other. Whereas in the case of A(I,J) = B+C; the E.P's domain is the entire statement. The E.P. should not return control at the '=' symbol!

(iii) Recognition of Error Conditions: Besides some terminal errors like illegal operator-operand sequence, recognition of an error condition may also depend on the special requests of the calling routine. For instance, in the example in (ii) ';' is a valid occurrence in A(I,J) = B+C; but not so in DO A(I,J);

(iv) Distinction between Pseudo Variable Reference and Built-in function reference: In PL 7044, there are several built-in-function names which may also be used as Pseudo variable names. The distinction is made again on the basis of context. For example, in COMPLX(A,B) = A+B+COMPLX(C,D);, the first reference COMPLX(A,B) is a Pseudo variable reference whereas COMPLX(C,D) is a built-in function reference.

(v) Recognition of scalar and aggregate expressions: For obvious reasons, processing of aggregate expression (i.e. expressions involving arrays and structures) need more work to be done than in case of scalar expressions. Now, if arrays and structures occur in the argument list of a procedure reference only, the expression will still remain scalar in nature. But if they occur in the argument list of some built-in functions, the whole expression can become aggregate in nature. For example, 'B+SIN(D)', where D was declared to be an array, will be considered to be of aggregate type, while 'B+P(D)', where P is a procedure by declaration, will be deemed to be of scalar type.

(vi) Analysis of the expression and generation of codes: In expressions, subexpressions are grouped together by means of parenthesis or by the well known rules of precedence (e.g. $a*b+c*d \equiv (a*b)+(c+d)$). Hence the expression processor must follow these rules in the course of its analysis, decomposition and generation of codes.

(vii) Meeting the requirements of different calling routines: The object code generated for the same expression in different situations, in general, will be different. This is because, different situations require different end results. For example, the occurrence of A+B, where both A and B are floating point variables, as format explicitors will necessiate the conversion of the end result into integer form. The occurrence of the same expression in a data list of an I/O command requires no conversion.

(viii) Ass[illegible] of Standard Side Effects: The result of the expression A+B+P(A), where P is a procedure by declaration and has the property of altering the value of A - such an effect is called the side effect - will be different in two different interpretations namely (A+B)+P(A) and A+(B+P(A)). This may cause the same program to give different results in two different implementations. Thus, in order to avoid any implementation dependancy in handling expressions, certain norms have to be introduced for tackling such situations. The norm that is universally accepted is that all codes relating to procedure references must take precedence over other codes in an expression. This will necessiate a final sorting of the processor output.

(ix) Separating the repetitive and non repetitive codes in case of aggregate expressions: For object code efficiency one would like to calculate the non-repetitive part of an expression once and for all outside the repetitive part. But in general, the repetitive and non repetitive parts will be interleaved in an expression. For this reason also, sorting of the code before finally outputting the code, is desirable.

The factors (i) thru (vii) can be taken care of by a single scan of the input by a single routine. But this will make the logic very involved. It was therefore decided to have a two pass scheme in the present implementation. The first scan takes care of factors (i), (ii), (iii), (iv) and (v) and is executed by a routine called the BUFFER HANDLER. The second scan is carried out by the ACTUAL ARITHMETIC PROCESSOR and accounts for factors (vi) and (vii). As has already been mentioned factors (viii) and (ix) are taken care of by the sorting routine.

In the present organisation, each calling routine will identify itself and its peculiar requirements by entering the buffer handler at a preassigned entry point. The buffer handler lays out the expression in a form that is recognisable by the actual arithmetic processor. The buffer handler then returns control to the calling routine after setting up relevant flags to indicate the type of the expression in question. It should be borne in mind that the calling routine calls the handler anticipating an expression in the particular situation in hand. It may turn out that there is no expression involved and this status is what is conveyed by the buffer handler when it returns control to the calling routine. Proceeding a step further, the calling routine examines the flags set-up by the buffer handler, and if the existence of an expression is indicated, a call to the actual arithmetic routine is put through. The arithmetic routine analyses the expression laid out in the fixed length buffer by the buffer handler, generates intermediate code, and calls the sorting routine. This in turns sorts the code according to predetermined priorities and returns control to the calling routine. This set up resulted in three distinct advantages (i) the modular structure of the organisation permitted parallel debugging of all the 3 routines and easier logic in coding, (ii) the calling routine was able to check the validity of the delimiter returned by the buffer handler, and thus the number of entry points were less than it would have been if each caller required a separate entry point- In other words, some of the entry points in the buffer handler could be used by more than one caller. This advantage accrued because the buffer handler returned control as soon it had done its job- and (iii) since both the buffer handler and arithmetic routine are called by the initial calling routine, certain amount of flexibility has resulted. The calling routine is in a better position to indicate its requirements to the actual arithmetic routine than the buffer handler. Thus the number of entry points in the arithmetic processor is also kept to the minimum.

As an illustration of all that has been discussed before, let us consider the following example:

```
PUT LIST ( ( ( (A+B) *C)**E DO I= 1,2,3) ) ;
         0 1 2 3  3'  2'              1' 0'
```

Here $(_1$ is used for bracketing the 'DO' whereas $(_2$ and $(_3$ are used for bracketing of expressions. In order to analyse this situation the I/O analyser calls the buffer handler only after consuming the 4 left parentheses. The buffer handler returns at $)_{3'}$ because of its inability to find a matching (. The I/O analyser supplies the matching left parenthesis and again calls the buffer handler which returns control at $)_{2'}$ for similar reasons. This continues till the buffer handler returns control after meeting the 'DO'. At this point the I/O analyser knows that there exists an expression consisting of 2 pairs of parenthesis and a 'DO' group. It can now call the arithmetic routine for analysis of the expression laid out by the buffer handler.

3.3 Input and Output:

3.3.1 System Limitations:

One of the most powerful features of PL/I is in the area of data transmission to and from the processor. As has been mentioned in Chapter I, this is also an area where maximum machine independance has been incorporated. The existing I/O software on the IBM 7044 is rather inadequate to handle the sophisticated requirement of PL/I I/O, the main reason being the I/O software is essentially designed to cater to the needs of FORTRAN like languages. Thus in deciding upon the features to be incorporated in PL 7044, one has to contend with limitations imposed by the system I/O software some of which are enunciated below:

(i) Of the three levels of IOCS available to the user only the second level-IOOP can accept requests for accessing randomly located records. Thus one has to write one's own buffering system (IOBS) if one wishes to incorporate non-sequential file processing.

(ii) The I/O routines which handle Fortran I/O statements are designed to handle records of data and are incapable of servicing stream oriented I/O commands.

(iii) File handling capabilities are relatively primitive. This is probably because the IOCS was written primarily to cater to FORTRAN like languages. For example no provision for checking validity of file .operations is made except on system input and output files.

(iv) Unlike Fortran, where a user may use a limited number of file names only (i.e. logical units 0 thru 7), PL/I is not restrictive in this respect. The linkage between file names(which are arbitrary) and the physical devices (or symbolic devices) is usually a duty performed by IOCS. No such operations are undertaken by the present system.

On the hardware side, absence of certain features like paging, backward reading tapes,makes it difficult or rules out certain features like BACKWARDS etc. Under the circumstances, two solutions presented themselves for consideration. First, one could revamp the entire I/O Software or atleast a part of it in order to suit the language requirements and second, one could write a number of supporting routines which might bridge the gap between what was required of an ideal I/O software package and what was available. The first solution, though better, in the long run, was too prohibitive in terms of man hours required, and one had to accept, perforce, the second solution although it suffered from a certain amount of clumsiness which is inevitable in any patchwork.

3.3.2 <u>File Handling</u>:

Files are declared just as any other variables by a DECLARE statement. They can also be declared by an OPEN statement. For a complete list of valid file attributes the reader is referred to Appendix G. At the time when the file is first used in an I/O statement, file declaration code is generated and outputted. Also the file is given a unique number depending on its lexicographic appearance in the program and thereafter the file is always referred to by this number. Since file names in PL/I can have a length more than 6 characters and since MAP language will not allow more than six characters, file declarations are made by forming a name such as 'PLF.XX' where XX denotes the file number. Thus a declaration in PL/I:

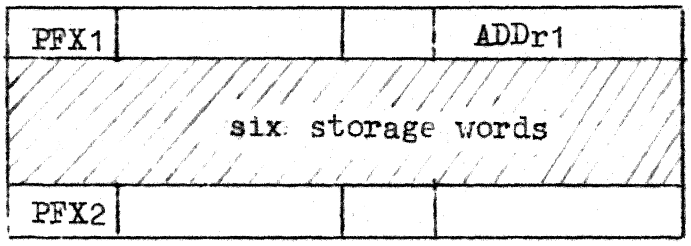
DECLARE MASTER FILE INPUT RECORD; will result in a MAP file declaration

```
PLF.00 FILE  U00,*,BLOCK=257,DOUBLE,LRL=256,TYPE3,RCT=1,
       ETC   REEL,EOF=REOFX.,ERR=RERRX.,EOR=REORX.
```

In the above example we notice that the file name 'MASTER' has been transformed into PLF.00 because it was the first file to have been declared.

Secondly we note that the linkage between the MAP file name and the symbolic device is also straight forward i.e. PLF.00 resides on U00. U00 is the abbreviation for utility unit having the symbolic name S.SU00. Notice that S.SU00 can be any device - disk, mag tape or any other device available in the system. The user must, of course, know that the order in which he declares his files is the order in which the utility units are assigned starting from S.SU00. If he declares more files then the number of utility units available, file declaration generation is suspended and an error message is given. We are disallowing multifile units because the user, in any case, cannot find out which of his files reside on a particular unit. As long as he restricts the number of files to less than or equal to the number of units available he can know for sure which files are on which unit.

As we have mentioned earlier, IOCS does not check the validity of operations on files. The file control block generated by the IBLDR does not contain enough information about files to cater to this need. It was, therefore, decided to maintain a file status block for each file, the format of which is given below:



PFX1	= PZE	for stream print files
	= PON	for stream input files
	= PTW	for stream output files
	= MZE	for record input files
	= MON	for record output files
	= MTW	for record update files
ADDr1	= link address of last file status block used in an I/O statement	
PFXA	== PZE	if this status block does not contain useful information
	= MZE	if the status block contains useful information.

The area shown under hatching is required to save certain information regarding status of the file. For example, if it is an input stream file, it is necessary to save the position (in words and characters) upto which the file has been processed. When the processing of a file is interrupted these pointers (word pointer and character pointer) are saved so that when processing is resumed on the same file the next character is made available. This is in contrast to FORTRAN file processing which is record oriented. In this mode of operation, every time the file processing is started, a new record is either read in or written out.

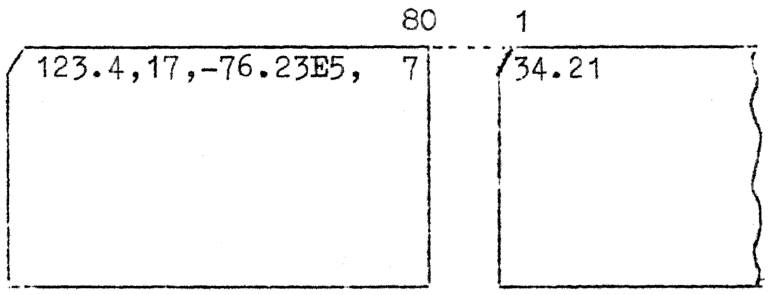
The link address Addr. 1 is necessary for flushing output buffers (if any of the files happen to be output files) at the end of the program. The exit monitor follows the chain so created and if an output file is indicated, and if some operation has been carried out on that file, than the corresponding buffers are truncated and written out. This ensures that no information is lost in the buffers.

3.3.3 Stream Oriented I/O:

The list directed, data directed and the edit directed I/O have all been implemented in PL 7044. The list directed I/O is essentially a free format facility intended to lighten the burden of I/O commands for the novice programmer. The data directed I/O is even more helpful to the newcomer in the sense that he can specify not only the values of the variables but also the names (consisting of valid identifiers) in the data stream. The edit directed I/O is meant for sophisticated editing of data items in accordance with format specification. Before we go on to describe the implementation of the three types of I/O commands it will be worthwhile to understand the mechanism by which record-oriented I/O routines of the system are made to appear as though they are stream oriented.

Let us specify, for simplicity, the card reader to be the input medium and the line printer to be our output medium. In case of FORTRAN, two successive read commands will cause two cards to be read, but in case of PL/I this is usually not the case.

Following figure shows two input data cards in which the values are punched in a free format (for a list directed input).



The first card is read in and the character and word pointers are initialized to point to the 1st column of the card. When a command to 'GET' a data item is issued, an input routine repeatedly calls a character fetching routine till it hits a non blank character. From this point onwards, the input routine will expect characters which are consistent with the definition of data constants.* When it encounters a comma or a blank it stops further scanning, does conversions if necessary for matching the target and source data items. Before quitting, this routine also updates the word and character pointer to point to the next character to be scanned. If the character pointer equals 80, it sets up a flag so the next time a command is given to 'GET' an item, the character fetching routine initiates a command to read a a new card, resets the various pointers and then starts the card scanning. Similarly in the case of a line printer enough information is kept in order to detect the end-of-line situation and initiate commands for outputting the just completed line and request IOBS for buffer space for the next line.

* See Appendix B.

The mechanism of list directed I/O has been explained above in connection with the conversion of record oriented I/O to stream oriented I/O. We shall, therefore, proceed with a brief description of DATA and EDIT - directed I/O.

Data directed I/O requires that the names of the variables participating in the I/O command be available at run time. It should be borne in mind, that in no other situation in PL/I is required to furnish variable names at execution time. Thus, it is necessary for one to create a table of the names, attributes and ~~x~~ the number and value of dimensions if any of the variable and maintain it at run time. Actually two such tables are created during the 2nd pass of the compilation. One is called the Compile Time Data Table whose capacity is limited to 500 words. The other table is the Run Time Data Table whose capacity is variable. Each entry in the Compile Time Data Table has the following . Format.

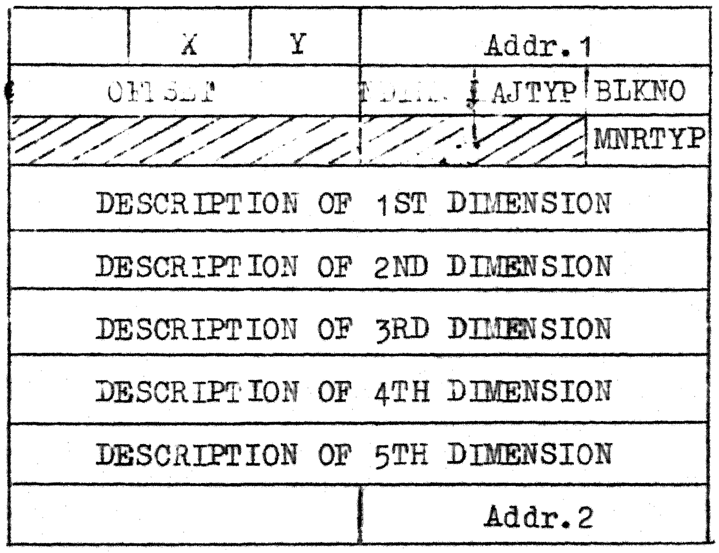

X
Y
Addr.1
OFFSET
TDIM
AJTYP
BLKNO
MNRTYP
DESCRIPTION OF 1ST DIMENSION
DESCRIPTION OF 2ND DIMENSION
DESCRIPTION OF 3RD DIMENSION
DESCRIPTION OF 4TH DIMENSION
DESCRIPTION OF 5TH DIMENSION
Addr.2

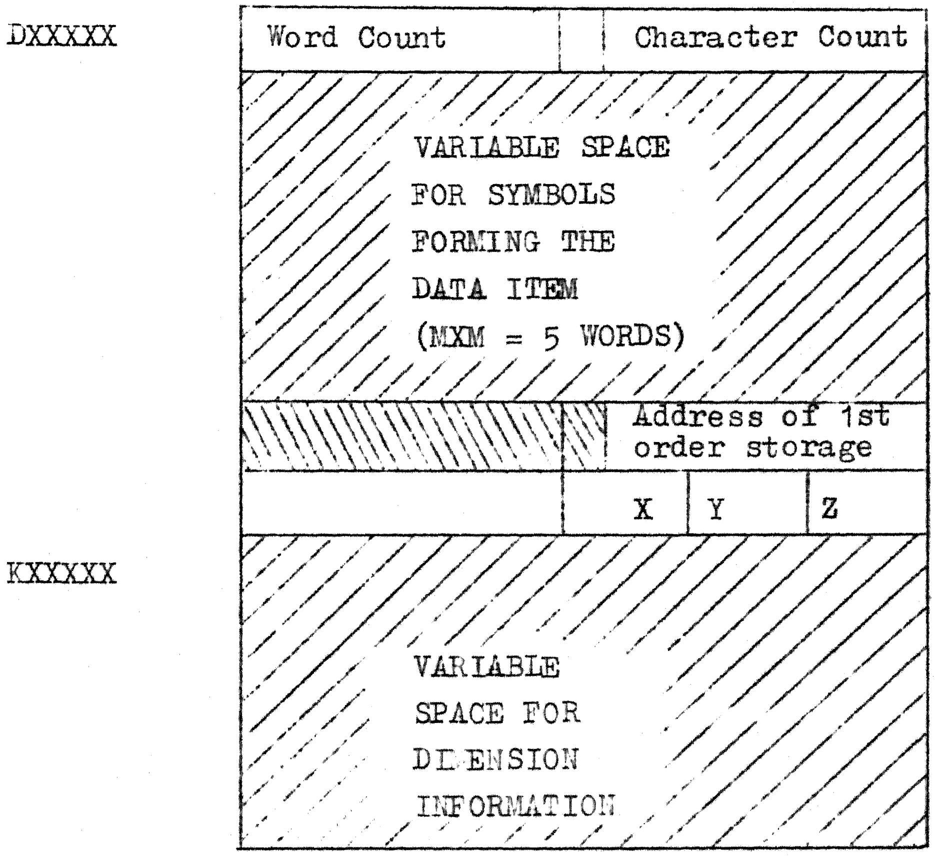
FIG. OC-3

where,

X	=	no. of characters in the name
Y	=	no. of words in the name
BLKNO	=	block serial number where the name is declared
MAJTYP	=	0 if it is a normal scalar variable
	=	1 if it is a format Scalar variable
	=	immaterial if variable is array
NDIMN	=	0 for a Scalar
	≠	0 for an array
Addr.1	=	address complement of the last entry in the table
Addr.2	=	address of corresponding entry in Run Time Table (in the form of a label number)

The purpose of a Compile Time Table is to minimise the number of entries in the Run Time Table and thus save space during execution time. Every variable in a GET/PUT DATA statement is matched against the existing entires in the Compile Time Table. If a match occurs, no Run Time entry for that variable is created, and only the address of the existing run time table entry (Addr.2 in Fig.OC-3) is transmitted to 3rd pass. If a match does not occur, a Run Time table entry and a Compile Time Table entry are both created and appropriate addresses passed on. It is interesting to note that the Compile Time Table does not contain the actual characters comprising the name of the variable, but only the attributes of the variable. This is because of the fact that each variable during compilation is completely characterised by their attributes alone, and there is no need to hold on to the symbols constituting the variable. In fact, one cannot say that symbols by themselves can uniquely define the variables, since the block structure of PL 7044 allows names to be non unique.

The format of the run time table entry is as follows:



DXXXXX
Word Count
Character Count
VARIABLE SPACE FOR SYMBOLS FORMING THE DATA ITEM (MXM = 5 WORDS)
Address of 1st order storage
X
Y
Z
KXXXXX
VARIABLE SPACE FOR DIMENSION INFORMATION

where,

X = No. of Dimensions

Y = Major Type

Z = Minor Type

In contrast to the Compile Time Table, the Run Time Table, holds the characters that form the symbol. This is as it should be, since in the input stream the only pertinent information available is the set of characters forming the name. Also, it should be noticed that whereas in the Compile Time Table each entry has a fixed number of words (=9), in the Run Time Table each entry has a variable format so as to optimise the number of locations required. Another difference is in the access mechanism of the two table. In the C.T.T. one accesses the entries (for searching) thru a linear chain, whereas R.T.T. is always examined selectively by the run time routines on the basis of the addresses provided to them by the translated code. In the figure depicting R.T.T. we have two labels per entry (if it is a dimensioned entry) or one label if it is a non dimensioned entry. The first label gives the address of the starting point of each entry while the second label gives the address of where the dimension information starts. Thus, for example, in,

```
GET DATA (A, B, C);
```

if A and B are 4-dimensioned variables and C is a scalar the following code is generated,

```
TSX  R.DATA,4
PZE  3
MZE  D00001,,K00001
MZE  D00002,,K00002
PZE  D00003
```

Thus, at execution time, enough information is made available for proper implementation of the GET/PUT DATA facility.

We shall now briefly touch upon the edit-directed I/O in PL 7044. Edit directed I/Os are always provided with format specifications. Format specification can either be given immediately after the closing of the data list, in which case they are called immediate format items or they may be given separately with a label prefix in which case they are called Remote Format specifications.

In so far as the implementation of the edit directed I/O instruction is concerned, there is not much difference between the two. What is of interest to us is the linkage between the format item and the data list. A cue has been taken from the manner in which Fortran Compiler does the linkage.The format is separately translated into a set of calls to various supporting routines as in Fortran, although the translation procedure itself is a lot more syntax and semantics ridden in case of PL/I formats than in the case of FORTRAN. Also it is quite likely that Format lists will be inter leaved with arithmetic processor's coding in order to calculate expressions used as Format explicitors. The list of variables occuring in the data list is separately translated into a set of calls to a linkage routine which then sets up the connection between the data list item and the format list items. The basic strategy has been borrowed from the Fortran Compiler, but a variety of extensions have been incorporated.

3.3.4 Record-Oriented I/O:

The translation of record oriented I/O has been made relatively simple in the absence of the BASED class of variables for reasons mentioned earlier. No special techniques had to be used for the implementation of record oriented I/O other than the validity checking operations that need to be performed before executing Record-oriented I/O commands also.

3.4 Concluding Remarks:

In this chapter, we have tried to highlight, some of the techniques evolved to implement PL/I on IBM 7044.

The introductory nature of the chapter prevented us from going into too many details. However, if the interest of the reader has been sufficiently aroused to read further, the purpose of this chapter would have been amply fulfilled.

CHAPTER III

THE LEXICAL ANALYSER AND ASSOCIATED ROUTINES

1. Introduction:

The lexical analyser is, in any assembler/compiler construction, the first routine to work on the input source text and is used for collecting, lexical primitives and assigning them tokens which are recognisable by higher level routines. The amount of work done by the lexical analyser (to be herein after called the lexicon) varies from compiler to compilerand from language to language. For example, for a high level language like PL/I the lexicon may have to do considerable amount of work in order to separate the lexical units, whereas for an assembly level language, the lexicon can be small and relatively unsophisticated. Also in the light of our experiences, it would seem that the greater the number of passes the sharper the dividing line between lexicon and other routines. This is because in a compiler having one or say two passes, the lexicon may have to take upon itself some amount of syntax analysis also, resulting in a merging of the two in some areas. However, if the number of passes is increased, then lesser and lesser work is done at the lexical level while more work is assigned to the higher level routines.

The lexicon for the PL 7044 scans the input source text and separates it into 29 different lexical units. Of these 6 are in the category of operands while the remaining are in the category of operators and delimiters. These are listed in Appendix A2. In PL 7044 the lexicon serves as a subordinate to the First Pass Processor. The lexicon has no provision for retracing or rescanning any part of the text. It has, however, a limited lookahead feature which is useful in certain situations.

2. Notations:

In the description that follows, the notations to be followed are,

LEXBUF	:	22 word buffer for collecting lexical units.
CRDBUF	:	72 word buffer for storing card column 1 thru 72.
CPOINT	:	Card Column Pointer
COUNT	:	Character Count of the lexical unit.
WCOUNT	:	Word count of the lexical unit.
FCOUNT	:	Count of characters after decimal point.
ECOUNT	:	Count of characters after exponent E.
TYPE 1	:	Lexical Type code when Lexin is used.
TYPE 2	:	Lexical Type code when Lxpost is used.
CODE	:	Keyword code (returned by search routine)
BCOUNT	:	Count of blanks required for padding.

The following entry points are provided in the lexicon,

LEXIN :	normal entry point
LXPOST :	lookahead entry point.

The lexicon requires the following routines as supporting routines,

ICV. :	BCD-Integer Conversion Routine.
FCV. :	BCD-floating point conversion routine.
ECV. :	BCD-floating point (with E) conversion routine.
ST. :	BCD-string conversion routine.
OCT.1 :	BCD-binary integer conversion routine.
OCT.2 :	BCD-floating binary conversion routine.
FIXTAB :	Part tree-Part binary keyword search routine.

The lexicon calls the following System /library routines.

JOBIN :	input editor for inputting source text.
JOBOU :	output editor for outputting source text.
S.XDVA :	DECIMAL Conversion of Address Part of ACC.

3. Description of Lexicon:

When a call is made for the first time, a card from the input file is read by a call to JOBIN and the first 72 columns of the card are arranged in the 72 words of the card buffer CRDBUF. For all subsequent calls to the lexicon, this part of the coding is skipped and only when the end of card is sensed, the next card is read in. The various information counters are initialized and the next character, as indicated by the character pointer CPOINT, is accessed. A jump on character is taken and this leads to a number of self contained routines described later in this section. The end-of-card is sensed as follows: The 73rd word in the CROBUF area contains the number $(64)_{10}$. Now in IBM 7044 there are 64 characters available and coded as numbers 0 thru 63. These codes, called '9-code' in IBM terminology are listed in Principles of Operation - IBM 7044. Thus when the card pointer indicates the 73rd word, the jump on character will transfer control to just outside the table used for the jump. At this point, instructions are provided to read the card. This is given by the transfer table given below:

```
        LXA  CPOINT,1
        TRA  TABLE,1
TABLE   BSS  0
        TRA  ... →  0    0th location from TABLE
        TRA  ... →       1st location from TABLE
        .                 .
        .                 .
        .                 .
        TRA  ... →       63rd location from TABLE
        READ ... →       64th location having a read-a-
                         card macro instruction.
```

Figure:LX-1

In all the following routines, the end-of-card situation is sensed in a similar manner, and this situation also signals the end of the lexical unit being collected. This means no lexical unit can be split on two cards. The exceptions to this rule are the comment field and the character and bit string constant.

3.1 <u>The Number Routine</u>:

This routine is entered when a jump is taken from locations 0 thru 9 from TABLE in Figure LX-1. This routine also makes use of a similar table for taking decisions but has different transfer points. The flow chart for this routine is given in Figure LX-3. As can be seen, from the diagram, the routine collects the digits in LEXBUF till it meets a non-digit character. If it meets a dot it first checks if a dot has already occurred, and if it has, then an error exit is taken. If the dot is occurring for the first time in the number, the corresponding flag is turned on and scanning is resumed. If it meets an 'E' it assumes a floating point number, and checks for the presence of a dot. When it encounters a 'B' during the scanning, it assumes a binary constant. When it encounters any other character, it terminates scanning, calls the appropriate conversion routine on the basis of the information about the constant collected, assigns the proper lexical type in TYPE 1 and returns. The number constant, in internal representation is available in LEXBUF. For the conversion routines to work properly, an integer number should not exceed $2^{27}-1$ (or nine digits), and a binary constant should not have more than 27 bits.

3.2 Identifier Routine:

This routine is entered when an alphabet A thru Z occurs as the first character of a lexical unit. This routine also makes use of a table similar to that shown in Figure LX-1 with different transfer points. Figure LX-4 shows the flow chart for this routine. This routine collects the alphabets A-Z and numbers 0 thru 9 constituting the identifier. An identifier may not be of length greater than 30 characters. When the scan meets a character other than those mentioned above, the scanning is terminated and if the number of characters is less than 12, a call to the keyword searching routine FIXTAB is made. This is because no keyword in PL 7044 is of length greater than 12. The key word search routine described in Section 4.1, returns after setting up an indicator indicating the result of the search. If the search is successful, then that key word's code and some relevant information is made available in 'CODE'. If the key word turns out to be a relational or logical operator formed of composite alphabetic characters, then it detected on return from the searching routine, and the appropriate type code is made available in TYPE 1. If the key word is not a logical or relational operator, then TYPE 1 is set accordingly and an exit from the routine is taken. If the search is unsuccessful, the type code for simple identifier is made available in TYPE 1 and control is returned to the caller. In all cases, keyword or no keyword, the identifier collected by this routine is made available in LEXBUF, the count of characters in COUNT and word count in WCOUNT and number of blanks in last word in BCOUNT.

3.3 The Quote Routine:

This routine is entered when a character constant as indicated by the first character quote (') is detected by the table in Figure LX-1. This routine also makes use of a table similar to that used in Figure LX-1 for decision making. If the character scanned, is any other than a quote, then it is collected in LEXBUF. If, however, the character is a quote, then the next character is checked to see if it is also a quote. If so, the second quote is collected in LEXBUF as a part of the character constant and scanning is continued. This is because the only way of including a quote character in a character constant is to precede it immediately by a second quote. Thus, for example, 'ABC''D''''FG'''' means the character constant ABC'D''FG''. Thus an odd number of consecutive quotes after the first quote, will always terminate the scan. On termination the next character is examined to see if it is a 'B' which signifies that the character string so collected is a bit string. Bit strings must not contain characters other than 0 or 1 e.g. '1110111011'B, and should not be og length greater than 27. A character constant should not be of length greater than 128. The character count, the word count and the character constant are available in COUNT, WCOUNT and LEXBUF, respectively.

If a bit string is indicated the routine ST.is called which converts it into internal form. This is then left justified and stored in LEXBUF. As usual all the information regarding the string is available in COUNT, TYPE 1 etc.

Figure LX-5 gives the flow chart for this routine.

3.4 The Comment Routine:

Control is transferred to the comment routine from the table in Figure LX-1 on recognition of the character '/'. At this point the next character is checked for a '*'. If it is not a star, then the appropriate type for a '/' operator is made available in TYPE 1 and control is transferred back to the caller. If, however, a '*' is the next character then the composite symbol '/*' is recognised as the start of a comment field. Comments may occur anywhere in the source text and may continue over several cards. Here also a table of 64 transfer points is made use of for arriving at a decision. When a '*' is encountered, the next character is checked for a '/'. If it is a '/', then the composite character '*/'is recognised which signals the end of the comment field. The flow chart for this routine is given in Figure LX-6.

In case the character encountered by the table in Figure LX-1 is a '+', '-', '(', ')', '*', or '=', then a simple assignment of type is made before returning to the caller.

In case a ',' character is encountered, the next character is tested to see if it is a '.'. In case it is, the composite character ',.' is recognised and an assignment of the code for a semicolon is made and control is transferred back to the caller. In case it is not a '.', assignment of the code for a comma delimiter is made and an exit from the routine is taken.

In case a '.' is recognised, the next character is scanned for another '.'. In case of a dot, the composite character '..' is recognised and the assignment of code for a colon is made before returning to the caller. If it is not a '.' then an attempt ismade to recognise a possible floating point number (e.g. .765). If the attempt succeeds, control is given to the number routine. If the attempt fails, an error exit is taken.

3.5 Lxpost Routine:

This is the lookahead routine mentioned earlier in the discussion. In certain situations the use of a dot character is valid only if the previous lexical type and the next lexical type are both identifiers. For example in A.B '.' is a valid lexical unit. But in case of A .B or A. B, '.' is not a valid character. Also in certain other situations, the calling routine would want to know in advance whether an operator/delimiter follows a particular operand. In order to handle such situations a lookahead routine was required.

The lookahead routine functions in the following manner. First TYPE 2 is initialized to zero. A table is then made use of for branching out depending on the nature of the next character. If the next character is anything other than a ',', '(', ')', '.' or a blank, then control is returned to the caller.

If the character is a dot and if the following contextual tests are successful, then TYPE 2 is set to indicate the code of a 'dot'. The tests are a) the previous type must be an identifier b) the character scanned just before the '.' must not be a blank and c) the next character after '.' should be an alphabet. If any of these checks fails then control is returned to the caller without updating the character pointer. An end-of-card indicator, as usual indicates termination of the lexical unit.

If the character is a ',', then depending on whether the next character is a '.' or not, TYPE 2 is assigned the code for either a comma or a semicolon.

If the character is a '(' or ')' TYPE 2 is assigned the appropriate code and control is returned to the caller. If the character is a blank, the blank flag is turned on and the next character is scanned. Figure LX-8 gives the flow chart for this routine.

3.6 <u>Error Recovery in Lexicon</u>:

The errors and the recovery action taken by the lexicon are as follows:

(i) 'Dollar' encountered in Column 1: This signals the end of the program. The dollar card is saved in the nucleus area S.SAVE and control is transferred to the routine 'DOLRMT' in First Pass Processor.

(ii) Illegal Character: Any character that cannot be recognised by the lexicon comes under this classification. A message to this effect is given, and the character is ignored and scanning is continued at the next character.

(iii) Length of Variable Exceeds 30: In this situation the length is truncated to 30 and the remaining characters forming the variable are skipped. A message is given to this effect.

(iv) Length of binary number exceeds 27. Action taken is similar to that in (iii). The number is truncated to 27 bits, remaining bits skipped over, and a message given to this effect.

(v) Length of Character Constant Exceeds 128. Action taken is similar to that in (111) and (iv). Character string is truncated, remaining part skipped over and an error message put out.

(vi) Illegal use of dot. As has been explained before, the dot can occur only under certain contexts. If these are violated, an error message is given to this effect and the occurrence of dot is ignored.

(vii) Exponent too long or Illegal: This error-areas when the exponent exceeds two characters or the character after E is not a ± or a decimal digit. After giving a message to that effect, the exponent is assumed to be zero .

(viii) Too may + or -: This error occurs when the exponent part of a floating point number contains more than one character for indicating the sign. e.g. 1.2 E + -1. In such cases only the first sign is taken into account. A relevant message is also given.

(ix) I correct formation of E-format Constant: This occurs when one tries to form a E format constant without a decimal point in the member. e.g. 1238E-12. Action taken is to treat the number as an integer and ignore the part E-12. A relevant error message is also given.

Figure LX-2 gives the overall flow chart of the LEXIN entry point of the lexical analyser. From the above discussion it is clear that the lexicon is programmed to give fast service to its caller. This explains the use of a large number of tables which take advantage of the particular BCD code of characters in IBM 7044. The lexicon was timed to read, process and write out cards at the rate of 120 cards per second. The read/write time for 120 cards on a tape-to-tape system turns out to be of the order of one third of a second. This gives an average processing speed of 180 cards per second. This is not a bad rate considering the amount of work undertaken by the lexicon. As an example the WATFOR Compiler's Preprocessor which does about half as much work, has a speed of about 330 statements per second* on IBM 7044.

* See WATFOR Documentation University of Waterloo.

4. Description of Supporting Routines:

4.1 Keyword Search Routine:

tThis routine is called by the identifier routine mentioned earlier. Its calling sequence is TSX FIXTBL,4.

The argument, LEXBUF is transmitted thru the control section 'LEX'. The search algorithm is a part-tree-part-binary approach which is the most optimum search technique for a fixed number of entries in a table . The schematic diagram for the search technique is given in Figure LX-7a. Each entry in the table consists of 4 words. The first two words contain the actual characters forming the keywords. No keyword in PL 7044 has a length of more than 12 characters. The 3rd word contains the code number of the keyword in the address part, its position in the symbol table in the decrement part and information regarding whether it can start a new statement in the sign bit. If the sign bit is negative, it indicates that the keyword in question can occur in the beginning of a statement e.g. DECLARE,GET,PUT etc. The fourth word of the entry contains pointers to other entries. The address part of this word points to the entry to be examined next, if the contents of LEXBUF and LEXBUF+1 is less than the contents of the first and second words of the entry. The decrement part of the word points to the entry be examined next if the contents of LEXBUF and LEXBUF+1 is greater than the contents of the Ist two words of the entry. If the address or decrement of the 4th word turns out to be zero, it indicates the end of the search along that particular branch. The variable 'FOUND' is set to minus if the search is successful or +ve if the search is unsuccessful. A search is terminated when either a match occurs (when it is termed successful) or when the end-of-search condition is encountered (when it is deemed to be unsuccessful). The initial tree which directs control on the basis of the first character of the keyword is incorporated in the decrement of the table shown in Fig. LX-1. Thus, this table is shared by the lexicon's decision mechanism and the searching routine. This makes for compact arrangement and is at the same time fast, although accessing a decrement field of a table takes slightly more time (3 cycles). Figure LX-7b shows the flow chart for this routine.

4.2 The Conversion Routines:

4.2.1 ICV.-BCD to Integer Conversion Routine:

This routine is called by the number routine, FCV, and indirectly by ECV. routine. It is used by all these routines to convert the character string in LEXBUF into internal integer representation. The input character must not be anything other than 0 thru 9. However, no checking is done in this routine,as this is performed by the number collecting routine in the lexicon. The number of characters should not exceed 9, as the integer constant in PL 7044 has a maximum value of $2^{27}-1 < 10^9$. The calling sequence of this routine is

```
TSX  ICV.,4
PZE  LEXBUF,,COUNT
```

The result is left in the accumulator. Fig. LX-9 gives the flow chart of the ICV. routine. The algorithm for conversion is heavily machine dependent and hence the flow chart contains, perforce, the actual instructions used. The octal constants used in the VLM (Variable Length Multiply) and VMA (Variable length Multiply and Accumulate) instructions are scale factors which are calculated for each position of the digit. As an illustration consider the scale factors for length equal to 2. The octal number 2000 is equivalent to the binary number 010 000 000 000. Assuming that we have a 1 in the units position of the BCD number in MQ (which is to be converted), we find that after the VLM instruction, the Accumulator will contain $(010000)_2$. Let us further assume that the tenths position in MQ contained a 1. Thus after execution of the VMA instruction, the ACC. will contain 00001 after four shifts neglecting the effect of addition involved in the VMA instruction. If we take into account the addition, the step-by-step transformation will occur as follows:

```
000 010 000      before executing VMA =ØC240,,4
001 011 000      after 1 addition and shifting
000 101 100      after 2 addition and shifting
000 010 110      after 3 addition and shifting
000 001 011      after 4 addition and shifting
```

The number ~~x~~after executing the VMA instruction is $(01011)_2$ which is equal to $(11)_{10}$. This is the number is BCD form that we started with.

The scale factors are thus calculated depending on the number of shifts that each has to undergo during the execution of these shift instructions.

4.2.2 <u>FCV.-BCD to Floating Point (with no Exponent Specification)</u>:

First the count is tested to find out whether there are more than 9 digits for conversion. If there are, only the first nine significant ones are taken for conversion. The initial conversion is done by the ICV. routine which converts the BCD form into internal binary form. After this, it is converted to floating point number and divided by the appropriate power of 10. For accessing the appropriate power of 10 in floating point form, a table of 79 entries containing the powers of 10 from 10^{-38} to 10^{+38} is made use of. The entries are arranged in ascending order of the power of 10. The result of the conversion is available in the accumulator. The calling sequence is

```
TSX  FCV., 4
PZE  LEXBUF
PZE  COUNT,,FCOUNT
```

The flow chart is shown in Figure IX-10.

4.2.3 <u>ECV.BCD-Floating Point (With Exponent Specified)</u>:

A call is first made to FCV. The result is saved. Then the exponent is examined for sign. If the exponent is negative the division by the appropriate power of 10 is carried out with the help of the table of the powers of 10 mentioned in Sec.4.2.2. If the sign is +ve then the appropriate multiplication is carried out. The calling sequence for this routine is, by ST. is positioned in the mantissa part, and then normalized. Fig. LX-12 gives the flow chart for this routine. The calling sequence TSX OCT.2,4. The arguments, as in the case of ST.are reached through the control section LEX.

5. The Intermediate Code Outputting Routine:

This routine is called by the First Pass Processor for the following reasons:

Calling sequence		Purpose
TSL HEADER	a)	To initialize 'JOBOU' routine for page heading, subheading and Page numbering.
	b)	To open and ready the two output files where output of Pass I will be saved for use by Pass II.
TSL OUTP.2	c)	To output the contents of 2 words from a particular location called 'BUFFER' in First Pass Processor routine on the 'Code' file.
TSL OUTP.1	d)	To output the contents of the accumulator preceded by a marker word of all 1's on the code file.
TSL OUTP.0	e)	To output the contents of the accumulator on the code file.
TSX OUTP.M PZE loc,,m	f)	To output the contents of m words from a location 'loc' on the code file.

Calling Sequence	Purpose
	g) To close the output 'Code' file and position it for Pass 2.
	h) To output information regarding no. of blocks, no. of DO's, block predecessor table, DO predecessor table etc. on the 'label'file.
TRA S.EXIT	i) To follow the chain made by Pass I Processor for the label information and output it for ~~f~~each block of the program on the 'label'file.
	j) To load the second pass via the system loader.
	k) To copy the code file and label punch on the System output file and System punch file if so desired during first pass.

In the above list certain terms and ideas need to be cleared. In items (c) thru (g) mention has been made of a 'Code' file and in items (i) thru (k) mention has been made of a 'label' file. The 'code' file consists of the translation of the source text carried out by the First Pass Processor with the aid of the Lexicon. The 'label' file consists of the label information for each block. Now the First Pass Processor has gathered the label information of each block as and when they occur and has prepared a chain for each block which is shown in Figure LX-13.

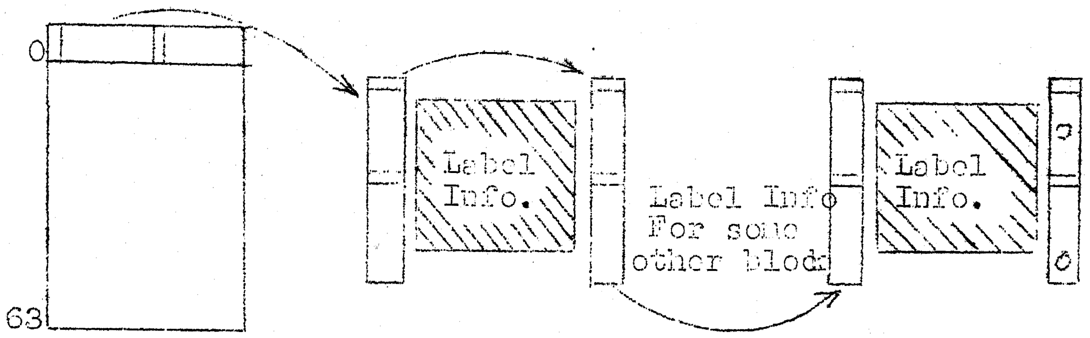


Figure LX-13: Chain for Block 0

The starting address of the chain for each block is stored in a tabular form. From then on, this routine follows the chain collecting the distributed label information and putting it on 'label' file till it hits a 0. This is the end of the chain. The shaded area in Fig. LX-13 gives the label information to be outputted.

The 'code' file fill and the 'label' file are named as PL IOO. and PL IO1. respectively and are both double bufferred. The output routine uses the IOBS level of IOCS and writes TYPE 3 standard length (= 256 words) records (except for the last record in each file) in binary mode. The units on which the files reside are given the intersystem reservation code of I17 and I18 in order to ensure their availability during subsequent phases.

Fig LX-2

LEXCON

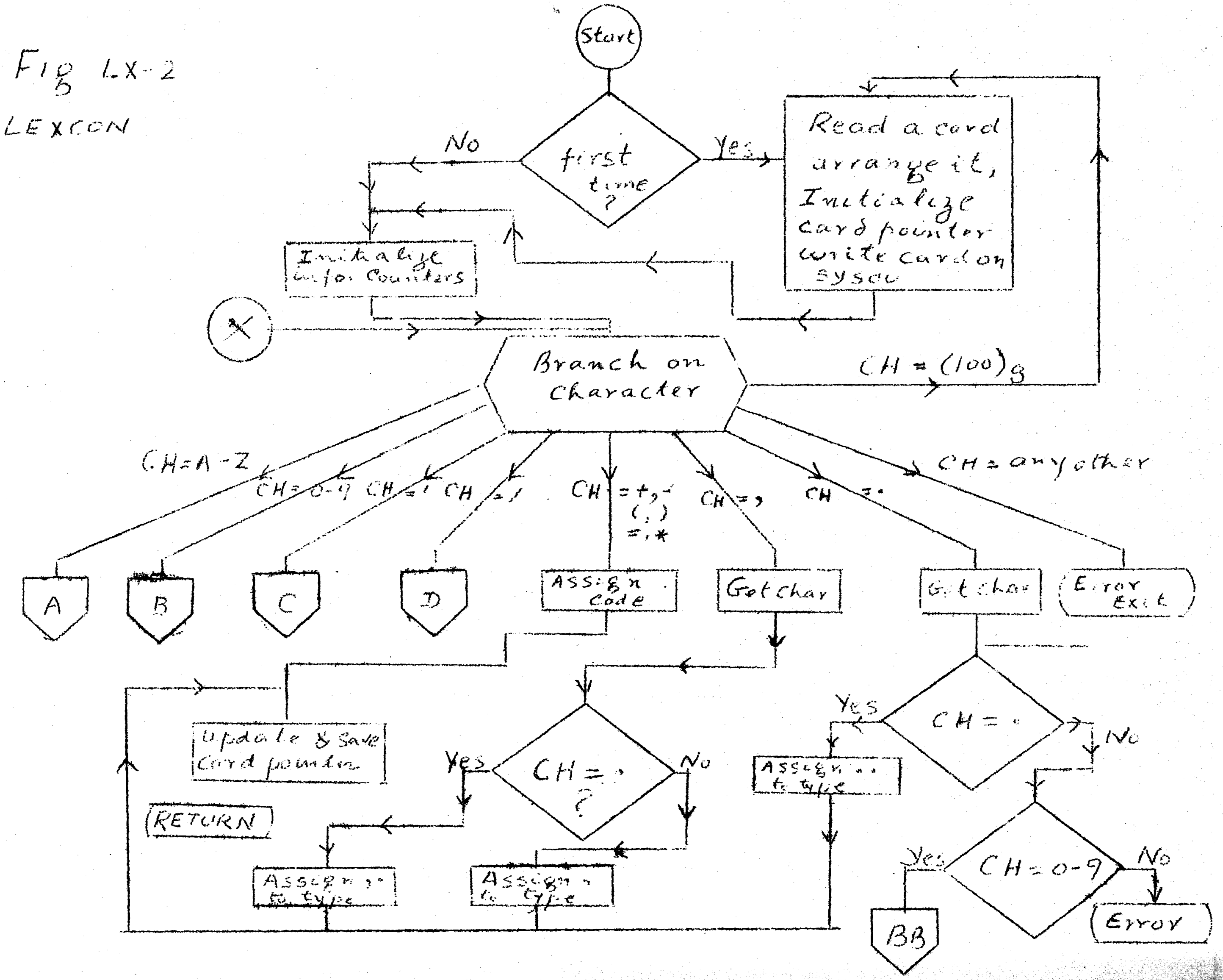


Fig LX-3
NUMBER ROUTINE

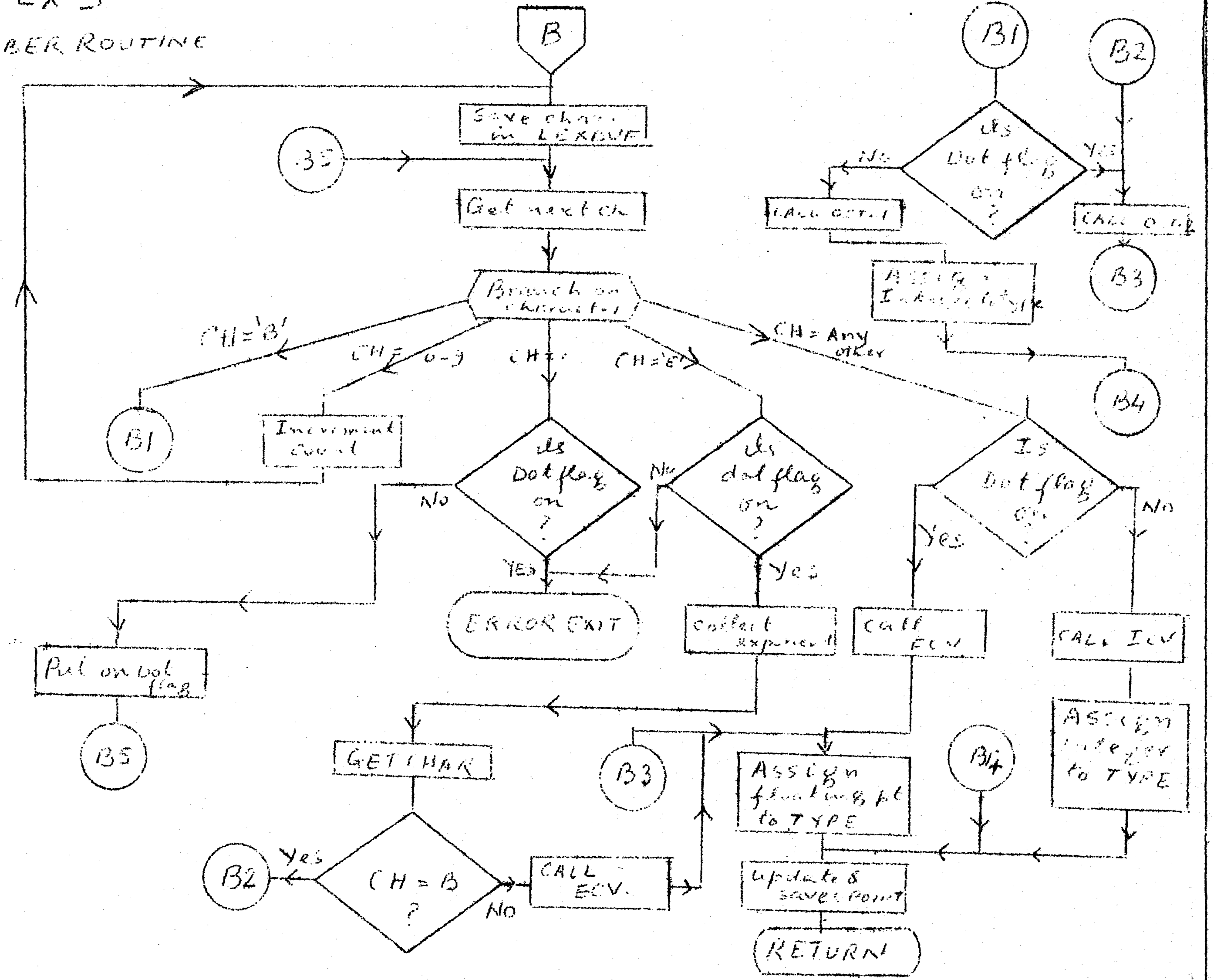


Fig LX-4
Identifier Routine

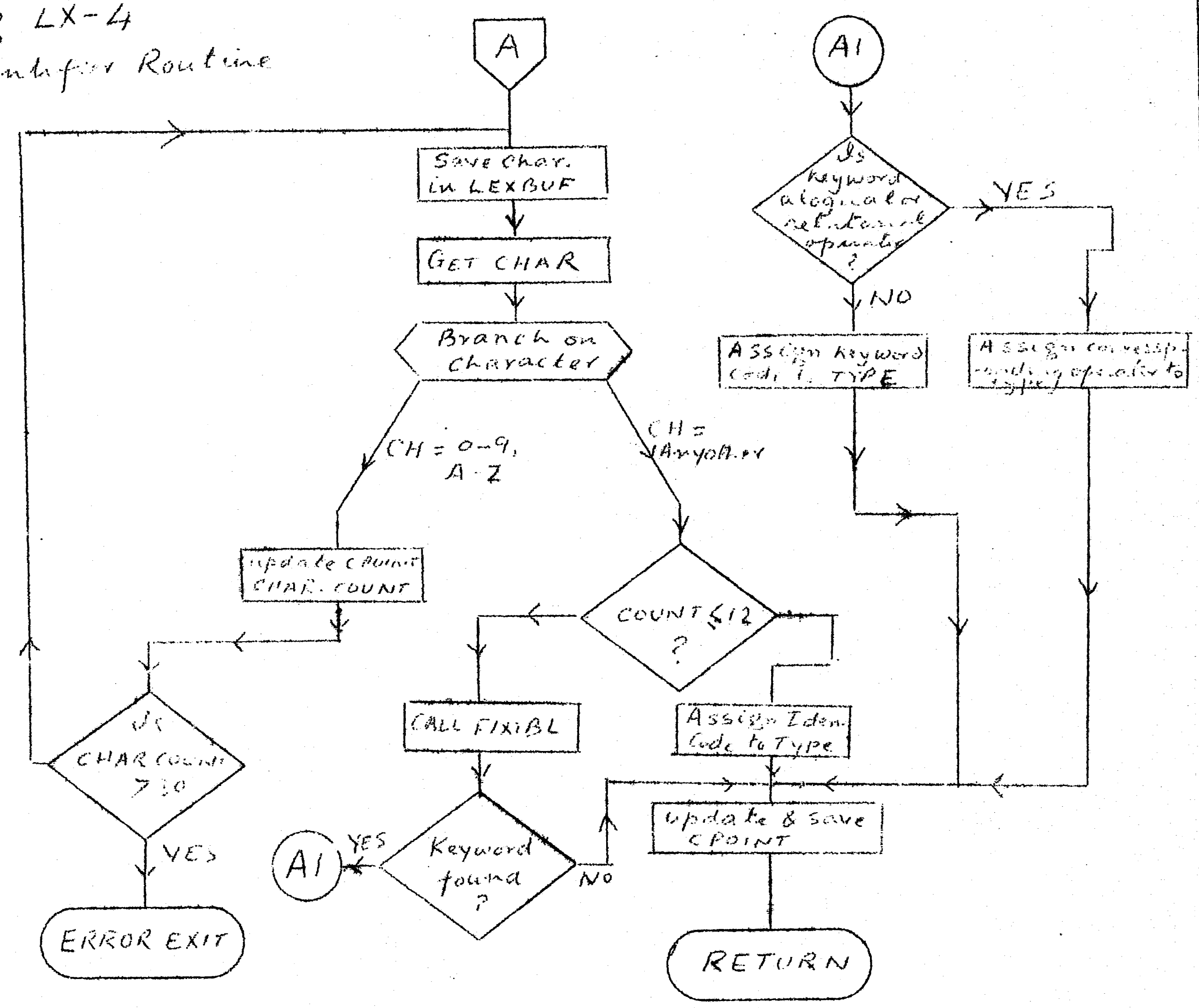
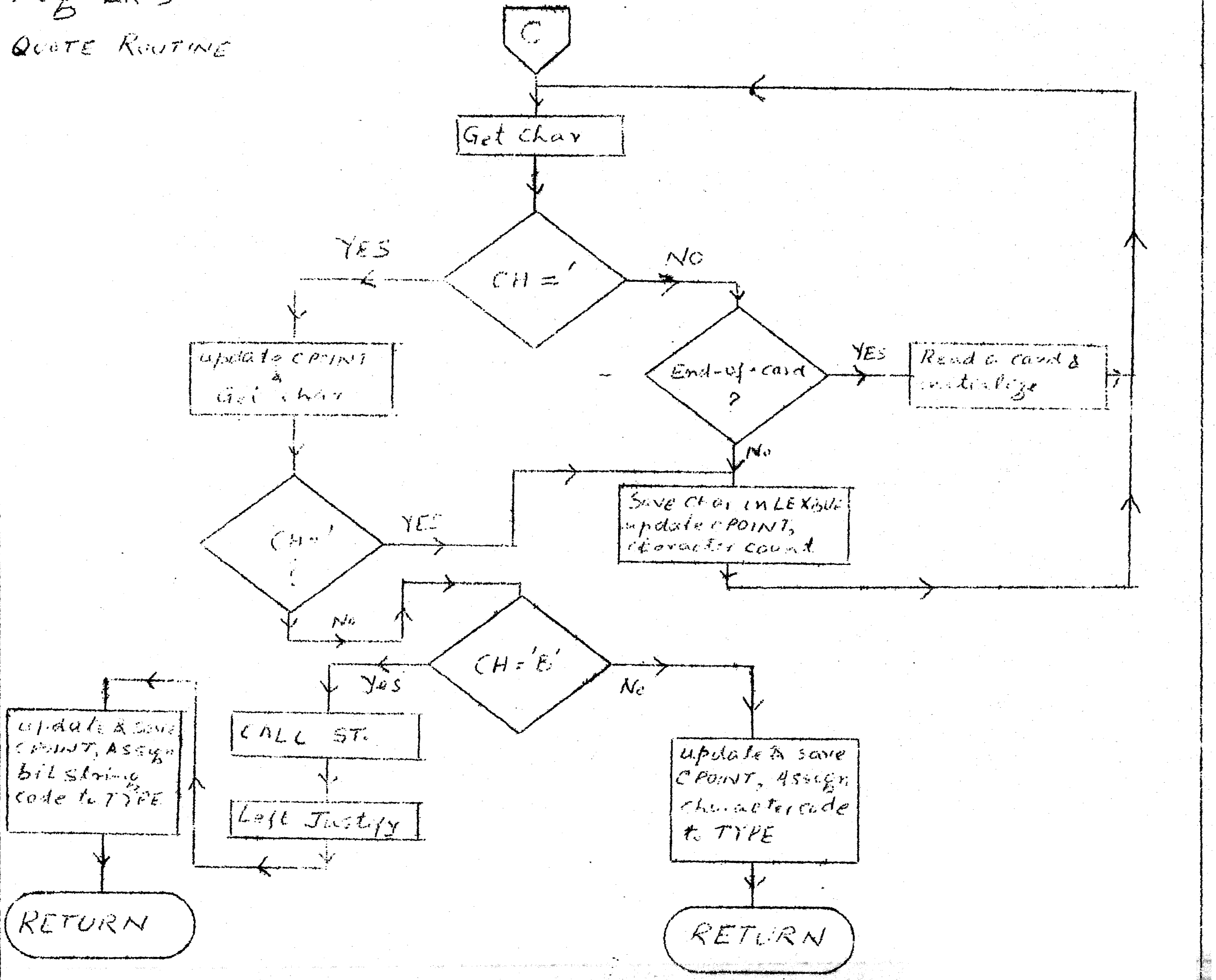


Fig LX-5

QUOTE ROUTINE


C
Get char
CH = '
YES
NO
update CPOINT & Get char
End-of-card ?
YES
Read a card & initialize
No
CH = ' ?
YES
Save char in LEXBUF update CPOINT, character count
No
CH = 'B'
Yes
No
CALL ST.
Left Justify
update & save CPOINT, Assign bit string code to TYPE
update & save CPOINT, Assign character code to TYPE
RETURN
RETURN

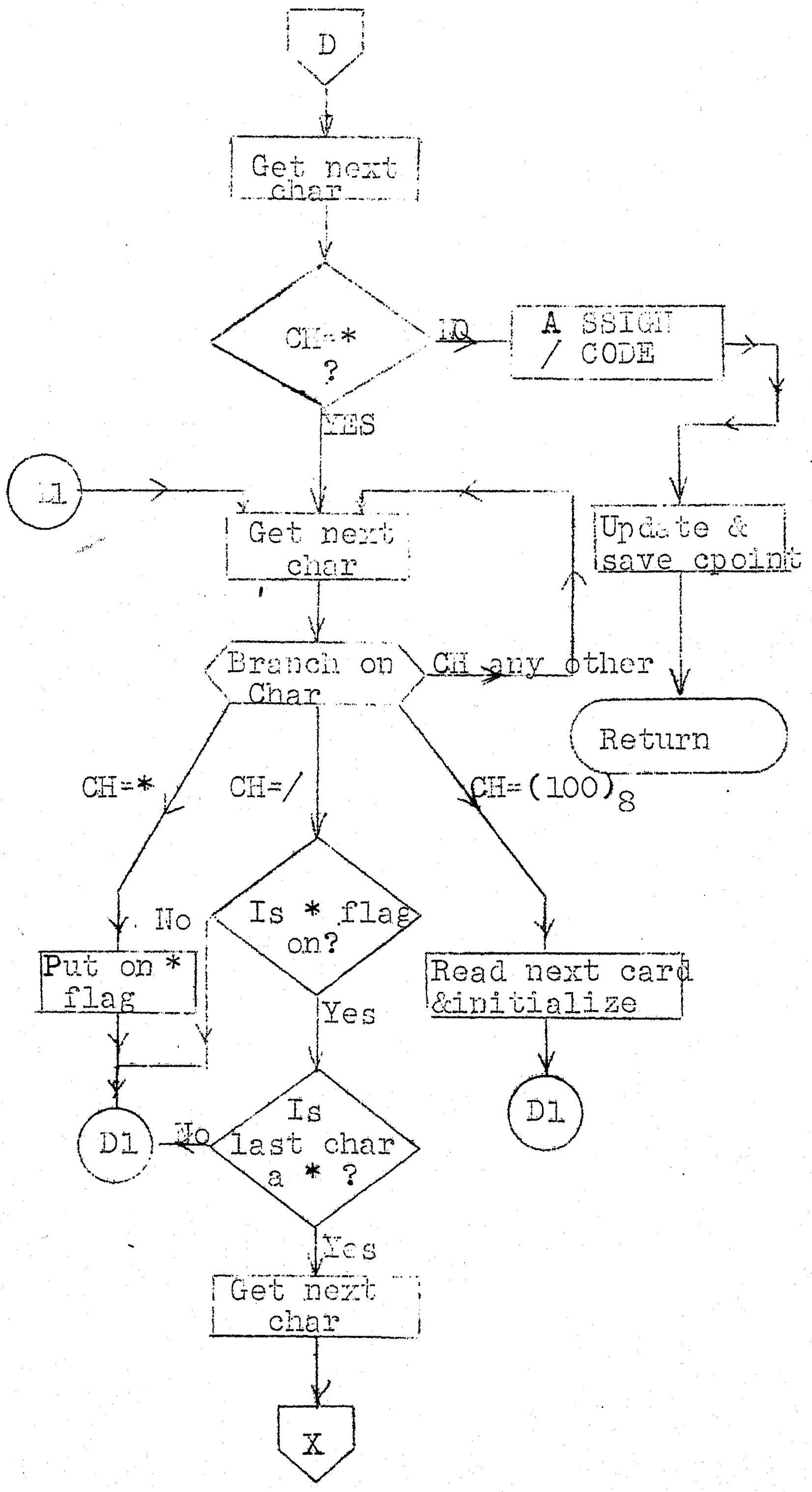


Fig LX-6: Flow Chart of Comment Routine

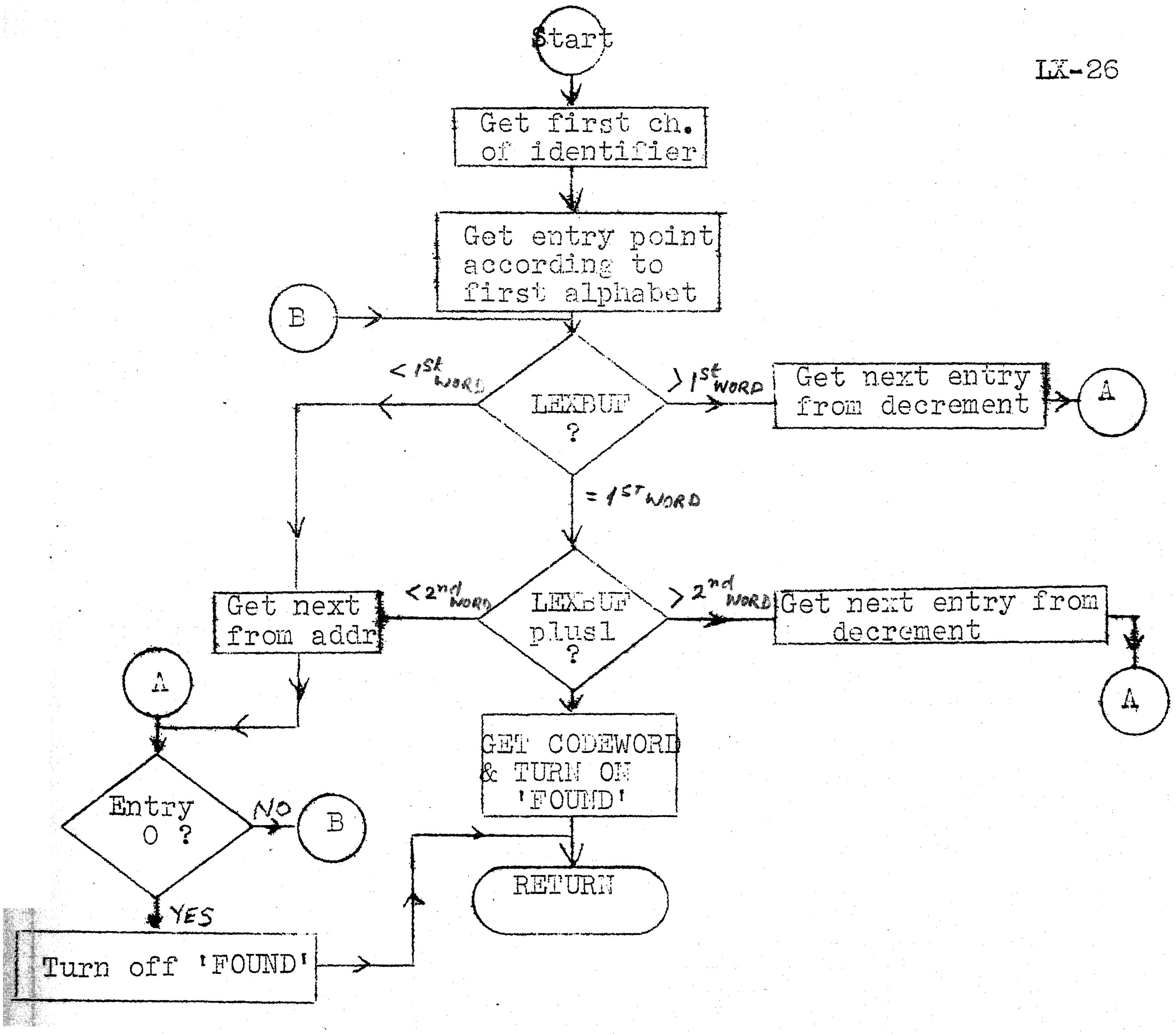

Start
Get first ch. of identifier
Get entry point according to first alphabet
B
LEXBUF ?
< 1st WORD
> 1st WORD
Get next entry from decrement
A
= 1st WORD
LEXBUF plus1 ?
< 2nd WORD
> 2nd WORD
Get next from addr
Get next entry from decrement
A
A
GET CODEWORD & TURN ON 'FOUND'
Entry 0 ?
NO
B
YES
Turn off 'FOUND'
RETURN

Fig LX-7a: Flow chart of Search Routine

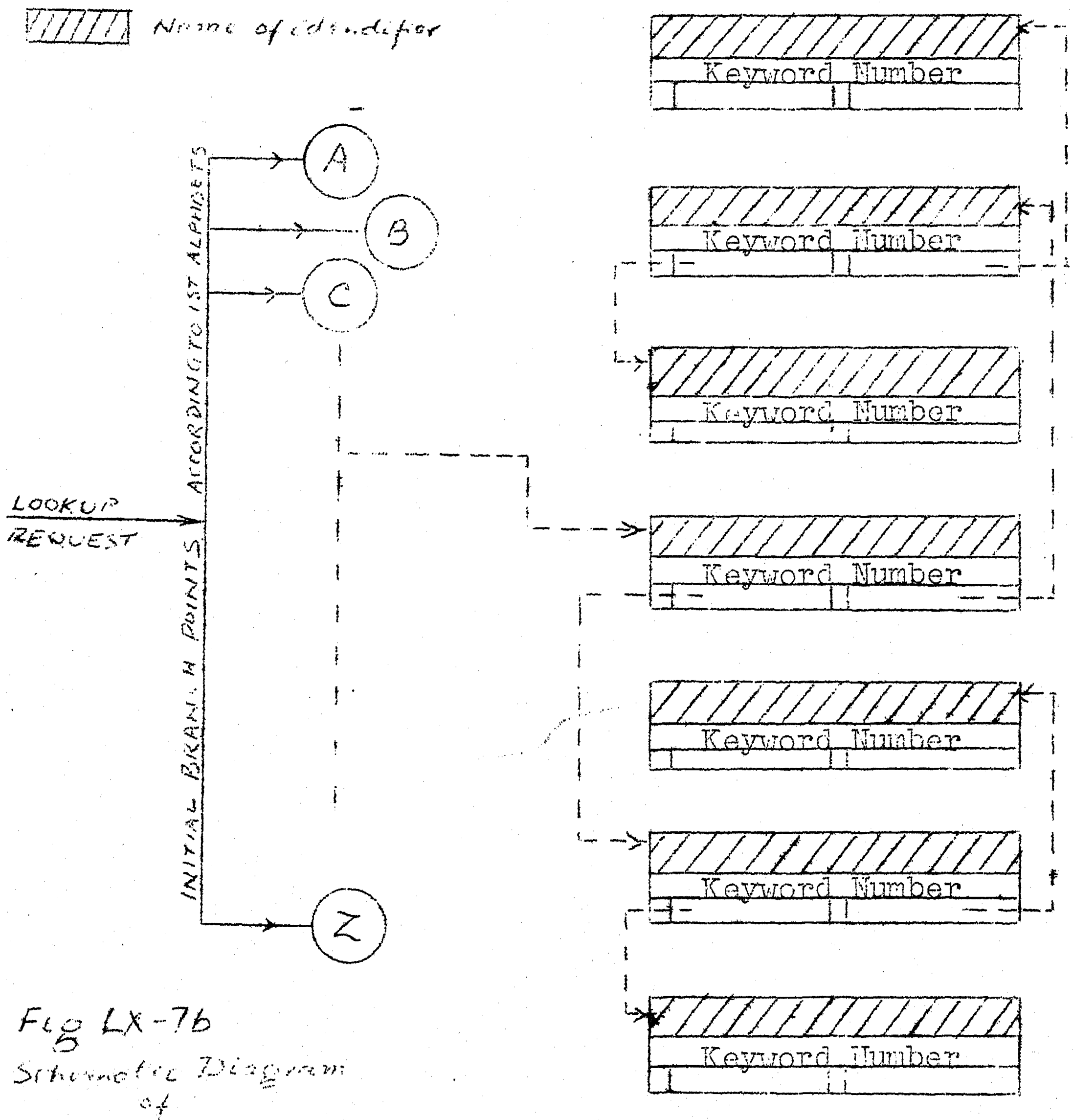
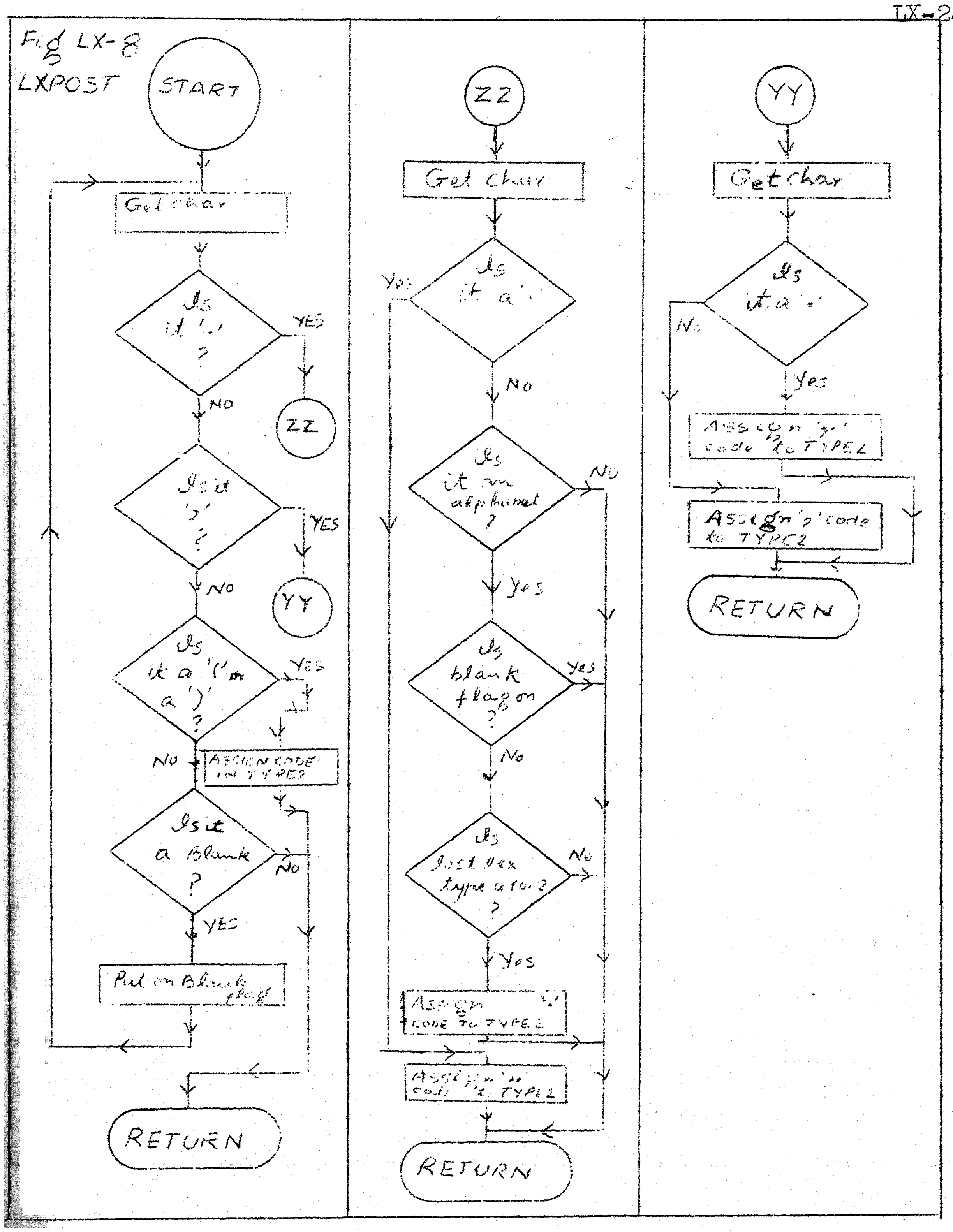


Fig LX-7b
Schematic Diagram
of
Searching Algorithm

Fig LX-8
LXPOST



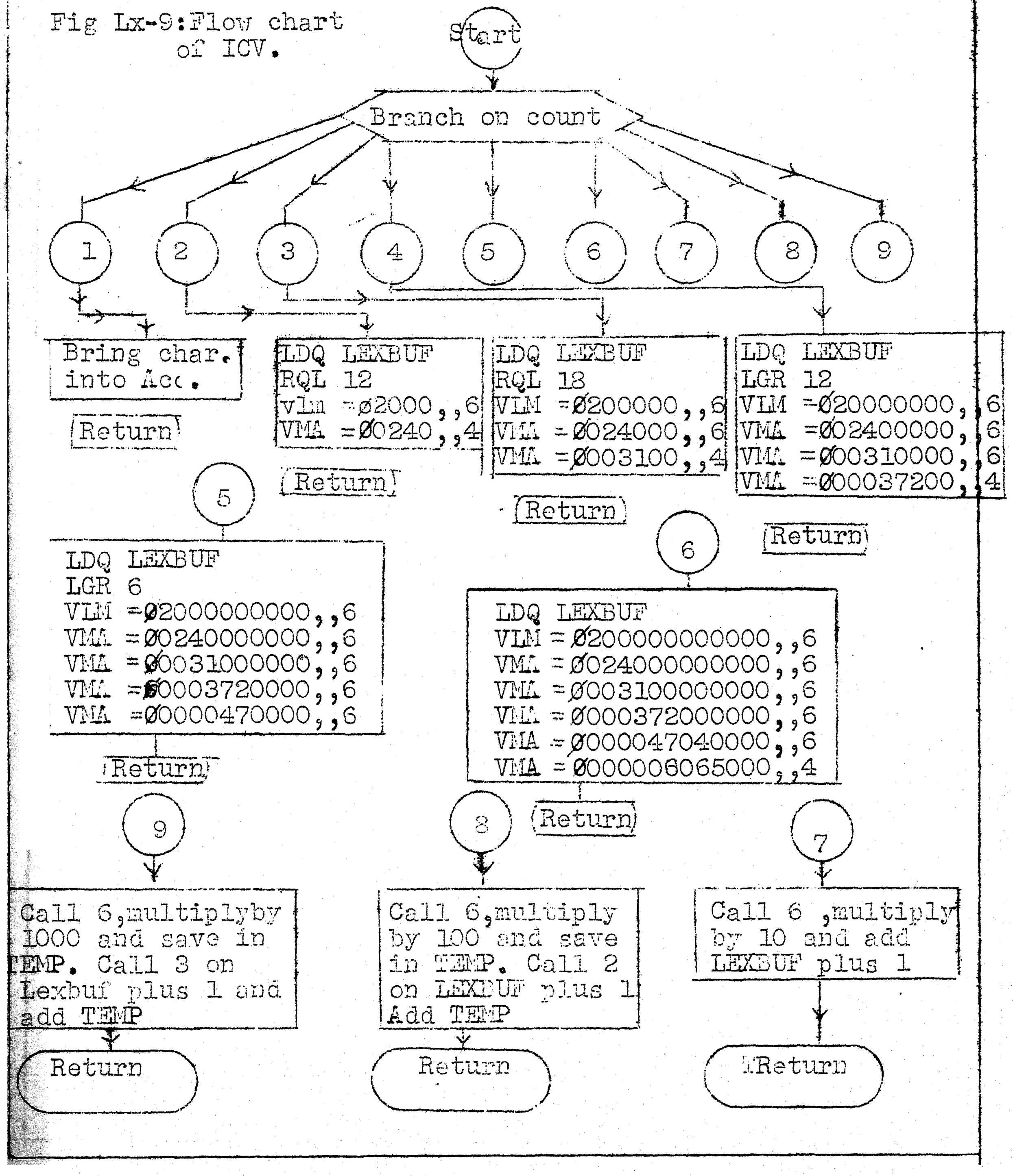


Fig Lx-9: Flow chart of ICV.

1:

Bring char. into Acc.

2:

```
LDQ LEXBUF
RQL 12
vlm =ø2000,,6
VMA =Ø0240,,4
```

3:

```
LDQ LEXBUF
RQL 18
VLM =Ø200000,,6
VMA =Ø024000,,6
VMA =Ø003100,,4
```

4:

```
LDQ LEXBUF
LGR 12
VLM =Ø20000000,,6
VMA =Ø02400000,,6
VMA =Ø00310000,,6
VMA =Ø00037200,,4
```

5:

```
LDQ LEXBUF
LGR 6
VLM =Ø2000000000,,6
VMA =Ø0240000000,,6
VMA =Ø0031000000,,6
VMA =Ø0003720000,,6
VMA =Ø0000470000,,6
```

6:

```
LDQ LEXBUF
VLM =Ø200000000000,,6
VMA =Ø024000000000,,6
VMA =Ø003100000000,,6
VMA =Ø000372000000,,6
VMA =Ø000047040000,,6
VMA =Ø000006065000,,4
```

9:

Call 6,multiplyby 1000 and save in TEMP. Call 3 on Lexbuf plus 1 and add TEMP

8:

Call 6,multiply by 100 and save in TEMP. Call 2 on LEXBUF plus 1 Add TEMP

7:

Call 6 ,multiply by 10 and add LEXBUF plus 1

FIG: LX-10a FCV.

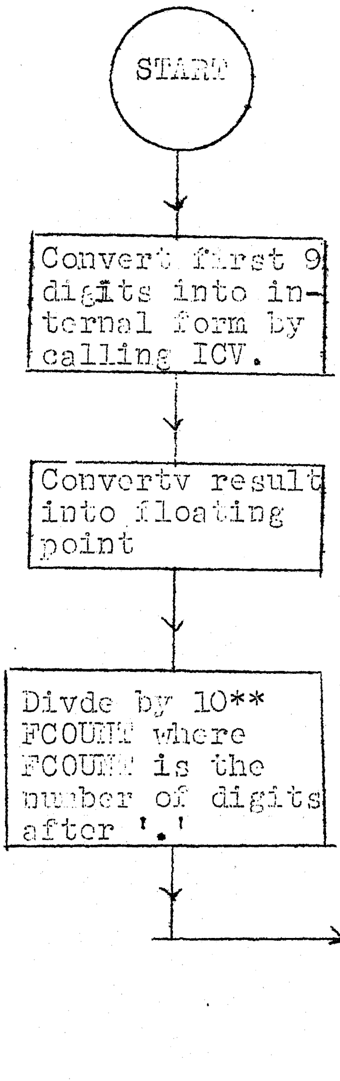

START
Convert first 9 digits into internal form by calling ICV.
Convertv result into floating point
Divde by 10** FCOUNT where FCOUNT is the number of digits after '.'

FIG: LX-10b ECV.

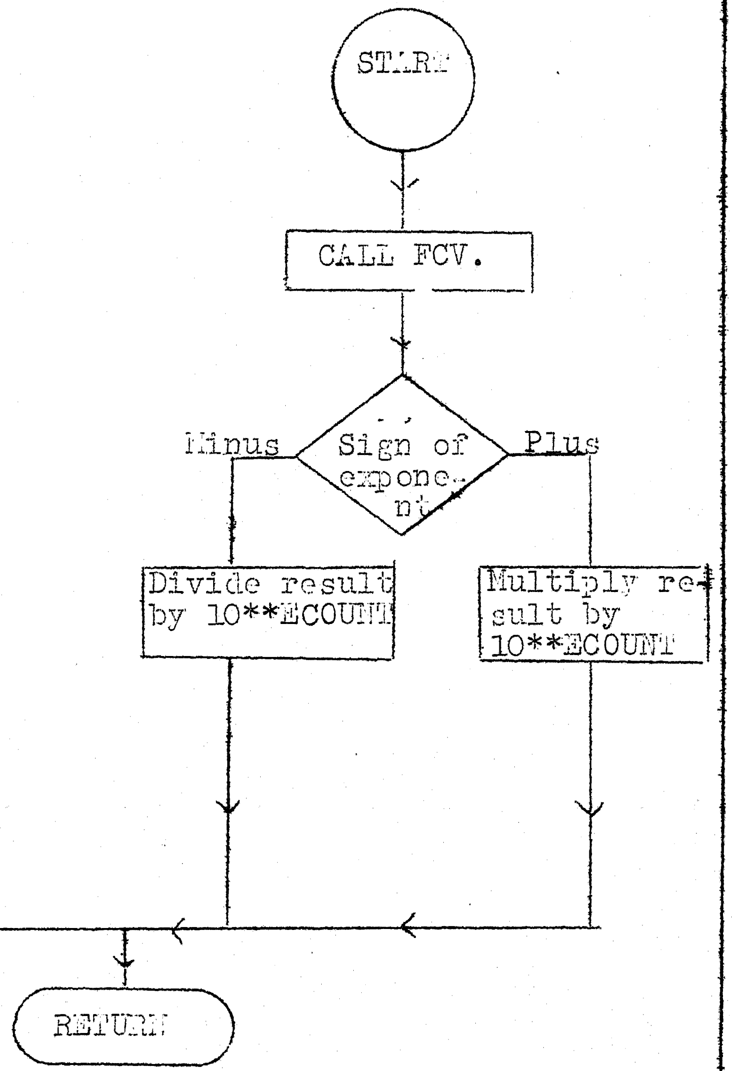
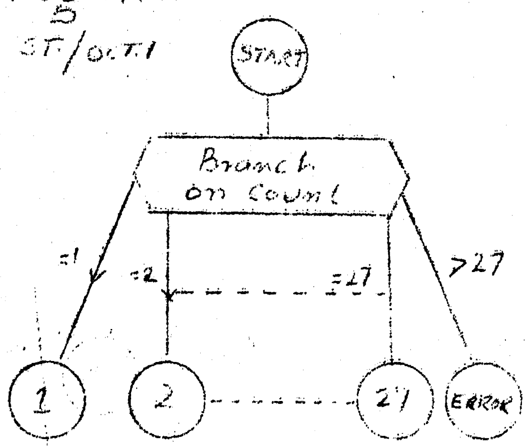
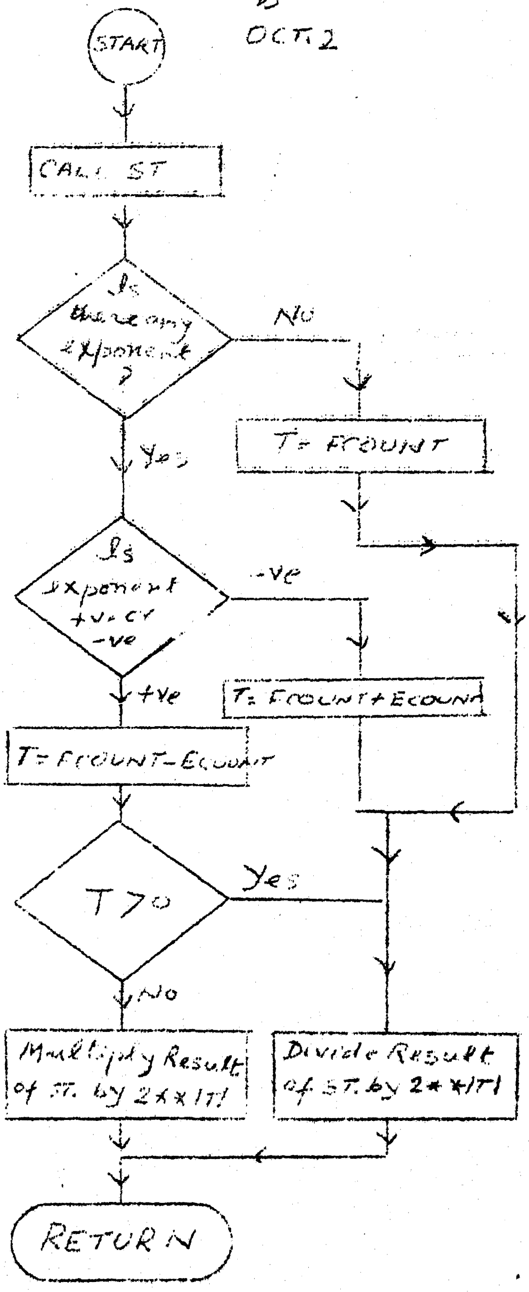


Fig LX-11
ST/OCT1



```
Code when Length = 1
  PCS   LEXBUF,,0

Code when Length = 2
  LDQ   LEXBUF
  RQL   12
  VLM   =Ø2000,,6
  VMA   =Ø40,,4

Code when Length = 3
  LDQ   LEXBUF
  RQL   18
  VLM   =Ø200000,,6
  VmA   =Ø4000,,6
  VMA   =Ø100,,4

Code when Length = 4
  LDQ   LEXBUF
  LGR   12
  VLM   =Ø20000000,,6
  VMA   =Ø400000,,6
  VMA   =Ø10000,,6
  VMA   =Ø200,,4

Code when Length = 5
  LDQ LEXBUF
  VGR 6
  VLM =Ø2000000000,,6
  VMA =Ø40000000,,6
  VMA =Ø1000000,,6
  VMA =Ø20000,,6
  VMA =Ø400,,4
```

Fig LX-12
OCT 2



CHAPTER IV

ANALYSIS OF I/O STATEMENTS AND THE DO GROUP

1. Introduction:

The PL 7044 I/O statements whose analysis and translation will be discussed in this chapter are as under:

a) GET statement,

b) PUT statement,

c) READ statement,

d) WRITE statement.

The GET and PUT statements are used with stream oriented files while the READ and WRITE statements are used with record oriented files. In addition, the DO grouping of data lists within a GET/PUT statements and the DO statement (outside the GET/PUT statements) have also been discussed in this Chapter.

The analysis routines, which are a part of the second pass, work in coordination with the first of the three arithmetic processing routines, namely the BUFFER HANDLER and the symbol table management routines. The code generating routines which are a part of the third pass use the intermediate code generated by the second pass analyser to produce MAP coding.

2. Syntax Considerations and Some Limitations:

As has been discussed in the first two chapters, PL 7044 has two types of I/O - the stream oriented I/O and the record oriented I/O.

In stream oriented I/O there are 3 types of transmission the list directed I/O, the data directed I/O and the edit directed I/O.

The most general form of a list directed I/O statement is as follows:

```
{GET/PUT} [FILE (< file name >)] [SKIP (<explicitor>)/
   LINE(<explicitor>)] [PAGE] LIST (data-list);
```

The general form for a data- or edit directed I/O statement is similar to that of a list directed I/O statement except that the keyword LIST will be replaced by DATA in one case and EDIT in the other. In the case of the edit-directed I/O, the data list is immediately followed by a format specification as shown below:

```
{GET/PUT} [FILE(<file name>)] [SKIP (<explicitor>)/
   LINE (<explicitor>)] [PAGE] EDIT (data list)
   (format-list)/(data list)(R(format designator))   ,
```

where format designator denotes a level constant or label variable prefixed to a format list. Also the definition of a data-list is different in the 3 cases and depends on whether the data list is used for input or output.

A data list is defined as,

```
element [,element] ...
```

An element, in the case of input is defined as follows:

(i) For a list or edit directed input an element may be a scalar variable, an array variable, a structure variable, or a repetitive specification involving any of these. Note that although PL/I allows pseudo variables they are not allowed in PL 7044.

(ii) For a data directed input an element may be an unsubscripted scalar variable, an array or its cross-section. Note that although PL/I allows structures as elements PL 7044 does not allow this. The reason for this restriction is discussed later in this chapter. Repetitive specifications are also not allowed because this leads to an ambiguous situation as explained under data-directed input analysis.

An element in the case of output is defined as follows:

(i) For a list or edit directed output an element may be a scalar expression, an array expression, a structure expression or a repetitive specification involving any of the above.

(ii) For a data directed output an element may be an subscripted scalar, an array or its cross-section. Although PL/I allows repetitive specifications in data directed output it is not allowed in PL 7044.

<u>Repetitive Specification</u>:

A repetitive specification in an input or output statement in PL 7044 has the following form:

```
        (element [,element] ... DO  Scalar Variable    =
                               Specification [,Specification] .....)
```

A specification is defined as,

```
                 ┌TO expression-2   [BY expression-3]┐
expression-1     │                                    │
                 └BY expression-3   [TO expression-2]┘
  [WHILE (expression-4)]
```

Each expression in the specification is a scalar expression. Repetitive specifications may be nested to a depth of 5 only although PL/I specifies no limit on the depth. If the DO-index variable is a string variable or an arithmetic variable of complex type, then the TO and BY options may not be used in the specification.

<u>File Usage</u>:

As specified in the general syntax rules , a file name may or may not appear in the I/O statement. If no file name appears in the statement, standard files (System Input file or System Output file) will be assumed. If however, a file name appears in the I/O statement, it must have been declared explicitly by a DECLARE statement. No contextual declarations are allowed. A file may have attributes added to it by means of the OPEN statement. But in every case the file usage must be preceded by its definition. If the attributes assigned to a file are not complete, default attributes will be assigned to make them complete. A list of attributes and default interpretations are given in Appendix G .

The PAGE option may be used only in an output command and on a file which has the PRINT attribute. Attempt to use it on any other file or in an input statement will raise an e error condition.

Certain Other Restrictions:

In a data- list- or edit-directed output command a character constant may not be used as an element although this is allowed in PL/I. For example,

PUT EDIT ('RESULT OF COMPUTATION ='); will raise an error condition during compilation.

Pseudo variables may not be used as a DO-index variable.

Record Oriented I/O Statements:

Two statements are allowed in PL 7044 for manipulating record oriented data. They are the READ and WRITE statements. The general syntax of a READ statement is,

```
READ FILE (file name) INTO (PLACE);
```

file name is the file from which the record is to be fetched and PLACE is the array where the record is to be stored in memory. Note that the transmission is literal and no conversion is made. The general format of WRITE statement is,

```
WRITE FILE (file name) FROM (PLACE);
```

This writes a record on the file given by 'file name' with the data contained in the array PLACE.

3. GET Statement Analyser:

The GET statement receives control from the second pass executive when the latter recognises the GET keyword. The organisation of the GET analyser is as follows: it has a common coding which analyses the part of the statement between the GET keyword and the keyword which classifies the statement as either list-or data- or edit-directed command. After recognition of the type of input command the program branches into three parts for further analysis. Before describing the analyser routine some notations are in order:

Notations:

FILDEC	This is a subroutine called by the main analyser for generating file cards and the file status blocks.
EX.PR	This flag is set by the buffer handler if it finds an expression in the statement.
H.DOBE	Entry point in buffer handler for handling the DO-index.
H.GET	Entry point in the buffer handler to handle a data item in the data list.
LBLCTR	Label counter used for keeping track of labels generated by the routine.
LPANY	Routine used for searching for excess left parentheses.

MAXVAL	Maximum value attained by LBLCTR.
LOOKUP	Routine to obtain symbol table information.
NAMSRH	Routine to search Compile Time Data Table.
Q.HELP	Routine called by the analyser when an error condition is raised.
STACK	Routine to save present count of excess left parentheses and present value of LBLCTR in stack.
UNSTCK	Routine to restore LPCONT and LBLCTR from the top of the stack.
LPCONT	Count of excess left parentheses.
XECUTV	Executive routine entry point.
STATUS	Gives the status of the stack (empty or non empty).
POINTR	Points to the free cell in stack.

3.1 The Common Coding:

The flow chart for the common part is shown in Fig. IO-1. This part first expects a file name to be used. If a file name does not appear immediately after the GET keyword, it assumes that the system input file is being used. If the system input file is being used for the first time, then a file status block is created and written out on the Work Unit on which the final MAP coding is to reside. The format of the file status block is given in Appendix K. For subsequent uses of the System input file, intermediate code is generated to direct the third pass to produce MAP coding which will initiate a call to the I/O supervisor. The functions of the I/O supervisor are described under the Chapter RUNTIME ROUTINES. The MAP code that is produced will be,

```
TSX  .IOSUP,4
PZE  S.FBIN,,S.FSBL.
```

If a file is explicitly mentioned, then the symbol table is looked up to examine the attributes of the file. The symbol table entry for a file name has a six word cell allotted to it just as for any other name. The information which is pertinent to our discussion here is available in the fifth and sixth word according to the following format.

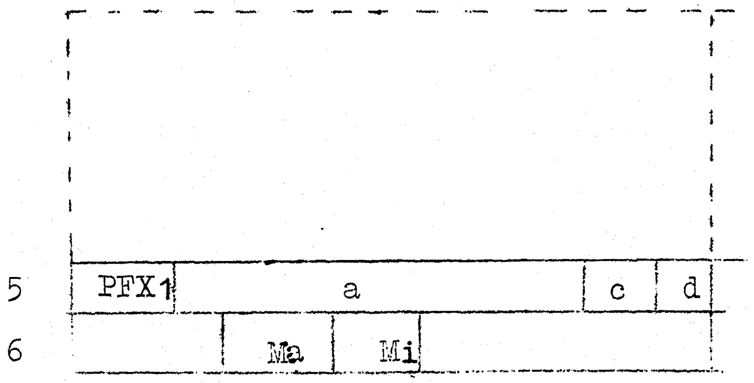


Fig. IO-2

where,

PFX1 = +ve for real files for which file declarations have not been generated.

= -ve for real files for which file declarations have been generated.

c	=	21 for a file which is declared in this block or is known in this block (such a file is called a real file).
	=	22 for a file which is a formal parameter (such a file will henceforth be called a formal file.
a	=	Offset in the first order storage area of the word allotted to a file if it is a formal file.
	=	File number if the file is real.
d	=	Block number of the word allotted to the file if it is a formal file.
Ma	=	Major family number for real files.
Mi	=	Minor family number for real files.

We first lookup the fifth character 'c' of the fifth word in order to check whether we are dealing with a real file or a formal file. If it is a formal file, the block number and offset of the word allotted to the file are passed on to the 3rd pass along with a code to indicate that a formal file description is being transmitted. The code generated by the 3rd pass for the situation would be as follows:

```
CLA  BS.XX+YYYYYYY
STO  *+2
TSX  .IOSUP,4
PZE  **
```

where XX is the block number transmitted and YYYYYY is the offset. BS.XX represents the starting address of the first order storage area of block XX and YYYYYY is the offset of the location w.r.t. BS.XX.

In case of a real file, the prefix of the fifth word is checked to determine whether the file declaration card and file status block have been generated. If they have been generated, then the code produced by the 3rd pass would be,

```
TSX  .IOSUP,4
PZE  PLF.XX,,F.XX
```

where XX is the file number obtained from the decrement of the fifth word of the symbol table entry for the file.

If the prefix of the fifth word is positive, it indicates that a file card and status block have to be generated. Before generating them, the attributes that have been collected by the DECLARE statement are examined for any apparent clash between the declaration and the usage. Note that although this check is performed at the time of file card generation, subsequent uses of the file will not be checked for validity. This is performed by the run time routines. For example, if a file is declared STREAM INPUT and used in GET statement for the first time, it will be okayed. But subsequently if it is used in a PUT statement no error will be given during compilation although during execution, the I/O supervisor will detect the error. The reason for checking attributes for the first time is to ensure that the file declaration card and the file status block are generated properly. The attributes of a file are available in coded form in the second and third characters of the sixth word. The former is known as the Major Family Name and the latter as the Minor Family Name.

Major Family = 0. This signifies that the file has been declared but without any attributes. The default mechanism specifies that a file declared without attributes is to be treated as a stream input file. Therefore, it is okayed and a file card is generated.

Major Family = 1. This signifies that either the attribute RECORD or STREAM has been assigned to the file. To decide which one of them has been actually assigned, the Minor Family Name is looked up. The Minor Family will be 4 for a STREAM attribute and 0 for a RECORD attribute. Thus, in the present case, Minor Family must be 4 else an error condition will be raised.

Major Family = 2. This signifies that one of the attributes INPUT/OUTPUT/UPDATE/PRINT has been declared. Further clarification is obtained by looking up the Minor Family Name which will be 0,1,2 or 3 corresponding to INPUT, OUTPUT, UPDATE or PRINT. For a file to be used in GET statement, the Minor Family should be 0 if Major Family is 2.

Major Family = 3. This signifies that one of the attributes from (STREAM/RECORD) and one out of (INPUT/OUTPUT/UPDATE/PRINT) have been assigned in the DECLARE statement. The Minor Family is then consulted for pinning down the attributes. The bits of information in the Minor Family name are obtained by ORing the information bits corresponding to the last two cases. e.g. if Minor Family is 5 (= 101_2) it means the attributes STREAM and OUTPUT have been assigned. For a file to occur in a GET statement Minor Family Name should be 4 (= 100_2) if Major Family Name is 3.

Major Family = 4. This signifies that the attribute assigned to the file is one of BUFFERRED/UNBUFFERRED. For a stream file, the attribute UNBUFFERRED is not permitted. The Minor Family Name will be zero for the BUFFERRED case and 8 for the UNBUFFERRED case. Thus, in the present case it can only be zero.

Major Family = 5. This signifies that two attributes, one form the group (STREAM/RECORD) and one from the group (INPUT/OUTPUT/UPDATE/PRINT) have been assigned to the file. Further clarification is obtained by looking up the Minor Family Name. The valid combination for a file in a GET statement is STREAM BUFFERRED and the corresponding value of the Minor Family Name will be 4.

Note in the case when Major Family is 1 or 4 or 5, nothing is mentioned about whether the file is to be used as input or output file. The default interpretation in such cases is that the file is an input file. Thus, if Major Family is 1,4 or 5 and it occurs in a PUT statement, an error condition will be raised.

Major Family = 6. This signifies that the attributes given is one from the set (INPUT/OUTPUT/UPDATE/PRINT) and one from the set (BUFFERRED/UNBUFFERRED). The valid combination for the file to occur in a GET statement is INPUT BUFFERRED and the corresponding value of the Minor Family is 0.

Note that in the case when Major Family is 2, 4 or 6 nothing is mentioned about whether the file is to be used in stream or record oriented commands. The default mechanism specifies that the file is to be treated as a stream oriented file. Thus, if Major Family is 2,4 or 6 and it occurs in a record I/O statement, an error condition will be raised.

Major Family = 7. This signifies that three attributes, one out of each of the groups (STREAM/RECORD), (INPUT/OUTPUT/UPDATE/PRINT) and (BUFFERRED/UNBUFFERRED) have been assigned to the file. The valid combination is STREAM INPUT BUFFERRED and the corresponding Minor Family Name is 4.

After the above validation checks are successful, the file declaration routine is called upon to generate the file card, and the file status block. The calling sequence for the file generating routine FILDEC is as follows:

```
TSX  FILDEC,4
VFD  6/X,12/YYY,12/ZZZ,6/W
PZE  m,,n
```

where X indicates the type of record (i.e. 1,2, or 3).

YYY	indicates length of block,	
ZZZ	indicates length of data record,	
W	indicates whether single buffering or double buffering is to be used,	
m	= 0	for stream print files,
	= 1	for stream input files,
	= 2	for stream output files,
	= 3	for record input files,
	= 4	for record output files,
	= 5	for record update files,
n	= file number.	

When FILDEC returns control, the sign bit of the fifth word in the symbol table entry is turned on to indicate that the file card has been generated for the file.

The input text is scanned for possible occurrence of the SKIP or LINE option. Only one of two commands may occur in a GET statement. The nature of the explicitor of the command (SKIP or LINE) is determined by a call to H.MISC which is an entry point in the buffer handler. If H.MISC indicates an expression then the actual arithmetic processor is called to output code for the expression. The GET analyser, then generates intermediate code which corresponds to the MAP code,

```
          TSX  IOHSC.,4
          PZE  <count>
    or    TSX  IOHLC.,4
          PZE  <count>
```

depending on whether a SKIP or a LINE command has been given. The common coding ends at this point. The next lexical unit is scanned for the type of input command and control is transferred to one of the three branches of the analyser.

3.2 The List Directed Input:

The 'modus operandi' of this part of the program is best illustrated by tracing the path taken by the analyser in parsing the following statement.

Example 1:

```
        GET LIST ((((A,B,C DO I=1 TO 10),D,E DO J=1 TO 20),
                          F,G,H DO K=30 TO 40), P,Q,R);
```

where A,B,C,D,E,F,G,H and P,Q,R may be scalar variables, arrays or structures array elements or DO groups. Each 'DO' group must be enclosed in a pair of parentheses. It must be noted that since the scanning of the data list is done from left to right and the 'DO' keyword appears at the end of the 'DO' group, the only way by which the analyser can recognise the occurrence of a 'DO' group is by the excess left parentheses it encounters at the beginning of a scan. Thus the first job of the analyser is to look for excess parentheses. This is done by the subroutine LPANY. If LPCONT is greater than 1, it means that there must be as many 'DO' groups as there are excess left parentheses (=LPCONT-1). In this case we find that there are 3 excess left parentheses corresponding to 3 DO groups. The start of the 3 'DO' groups are assigned 3 labels. The label number is kept track of by LBLCTR. After the assignment of labels, code is generated for transferring control to the outermost 'DO' group label. Thus the code generated so far will be,

```
              TRA M00000
       M00003 TRA **.
```

If we assume that A,B and C are simple scalars, then the following action takes place. The analyser calls H.GET in buffer handler for determining the nature of the data element. H.GET returns with the expression flag EX.PR off (since A is not an array element). Also the structure flag STRFLG is off since A is not a structure. The address of the symbol table is available in a preassigned location in the buffer. By looking up the symbol table entry, the analyser can obtain the attributes of the data element A. Similarly the attributes of B and C are looked up and the intermediate code generated corresponds to the MAP coding,

```
        TSX  G.LIST,4
        PFX1 A
        TSX  G.LIST,4
        PFX2 B
        TSX  G.LIST,4
        PFX3 C
```

Here PFX1, PFX2 and PFX3 are prefixes which carry information about the type of the variable.

At this point the analyser encounters a 'DO' key word. This indicates that the domain of the innermost 'DO' is over and that the coding of the 'DO' must start now. The intermediate code that is generated at this point corresponds to the MAP coding,

```
              TRA  M00003  (Signifies the end of domain of
                            the innermost 'DO' group)
       M00002 TRA  **
```

A call to H.DOBE is then made in order to determine the nature of the 'DO' index. If the EX.PR flag is on it indicates that the 'DO' xindex is an array element. Otherwise it indicates a scalar variable. In our case the index I is a scalar of type integer. The type is saved for future reference. Now the input text is scanned for a parameter. The parameter (or specification as it is called in PL/I) may be a scalar expression or a 'TO-BY' specification. The former is trivial but the latter case requires some explanation. The entry point H.MISC is invoked to determine the nature of the expression following the 'TO' keyword. If it turns out to be a constant or a scalar variable the expression processor is not called. If however, the occurrence of an expression is indicated, then the expression processor is called upon to generate coding for the expression. The result of the expression is made available in a temporary location at run time. The description of the temporary is what is returned by the expression processor. Similarly the BY expression is also handled. The description in each case is transmitted to the 3rd pass which then generates the actual code. If a 'WHILE' clause is also mentioned, then the expression processor is again called to generate appropriate code. In this case, however, two labels, one called the 'TRUE' label and the other called the 'FALSE' label, are generated. The 'TRUE' label is one where control will be transferred if the 'WHILE' clause is satisfied and the'FALSE' label is one where control is transferred if the 'WHILE' clause is not satisfied.

In the example at hand, for the inner most 'DO' only a 'TO' is involved. Thus the intermediate code for this will correspond to the following MAP code,

```
       CLA  =1           Initialize 'DO' index I
       STO  BS.XX+YYYYYY (Internal representation of I)
       CLA  =10          Set upper limit
       STO  TS.XX+ZZZZZZ (Internal representation of temporary)
       CLA  =1
       STO  TS.XX+WWWWWW Set default increment
X00001 CLA  TS.XX+ZZZZZZ Check if 'DO' index exceeds the upper
                         limit
       SUB  BS.XX+YYYYYY
       TMI  Y00001
       TSL  M00000       Call domain of 'DO'
       CLA  BS.XX+YYYYYY
       ADD  TS.XX+WWWWWW increment 'DO' index
       STO  BS.XX+YYYYYY
       TRA  X00001       Go back for more
Y00001 EQU  *
```

After generating the above code, further scanning is resumed. At this point let us examine the need for a stack in processing a 'DO' group within another 'DO' group. In Example 1, if we assumed that D is another 'DO' group enclosed in a pair of parentheses, it is clear that we have a 'DO' group within the domain of another 'DO'. In such cases we have to save the status of the outer 'DO' before we can enter the inner 'DO'.The information to be saved consists of the excess left parentheses 'LPCONT' at the point we encounter another 'DO' and the value of the label counter 'LBLCTR' which indicates the next label number that may be used. This saving is done by the routine STACK. It uses the decrement and address part of a word to ~~s~~ store the two counters LPCONT and LBLCTR. After returning from 'STACK' LPCONT is initialized to 1 and LBLCTR is left unchanged. We can now proceed with the inner DO's as we did in the beginning of the scan of the data list. After we emerge from the inner 'DO' groups (which is indicated by the right parenthesis) we perform an unstacking operation by calling the routine UNSTCK. This restores the value of LPCONT and LBLCTR that existed before we met the inner DO. The number of such nesting of DO's has to be limited because the stack in the routine has only finite number of words for holding information. In the present implementation this number is fixed at 5.

Let us resume our discussion of example -1. Let us assume that D and E are arrays of dimension 3 and 4 respectively and F is a structure. Now, array element addresses for D and E are calculated at run time only because, in general, the bounds of the dimensions are variables. Therefore, code must be generated to calculate the addresses at runs time. A supporting address calculating routine is called, which supplies the starting address of the array data area, the increment necessary for obtaining the address of the next element and the iteration count. Thus the job of the analyser is made relatively simple. It generates code to initiate a call to this routine (F.RNGE) and supplies it with the header word of the array and the three labels where the information calculated by F.RNGE is stored. In particular the MAP code generated will be,

```
TSX  F.RNGE,4
PZE  <header address>,,<label 1>
PZE  label 2,, label 3
```

where label 1 is the location where the starting address of the header will be stored, label 2 is the location in whose decrement part the increment will be stored and label 3 is the location in whose decrement part the upper limit of the elemental address is stored. Thus the three labels may be used in the following way,

```
label1  AXT  **,1
        SXA  *+2,1
        TSX  G.LIST,4
        PZE
label2  TXI  *+1,1,**
        SXA  label1,1
label3  TXL  label1,1,**
```

Cross sections of arrays may be treated in a similar fashion but for cross section of arrays, the arithmetic processor generates the relevant code. For array elements also the arithmetic processor is responsible for generating code which will calculate the element address and store it in appropriate locations.

In case of a structure (major or minor) like F, the buffer handler routine returns after turning on the structure flag STRFLG. The symbol table entry address is available in a pre-assigned location in the buffer. For a structure the nodes are connected through a chain. This chain will eventually lead to the base elements. Certain indicator are available in the symbol table entry of a node by which the analysing routine can know when it has hit a base element and when it has hit the end of the chain. For each of the base element of the structure a call to the run time routine G.LIST is made as was the case for simple scalar elements.

3.3 <u>The Coding of the DO Statement</u>:

Although the 'DO' statement is a separate logical entity by itself. the coding of 'DO' statement is very similar to the coding of the 'DO' group of the I/O statement. Hence it is most appropriate to discuss it at this point. In the case of the 'DO' statement, the specifications regarding 'DO' index, DO parameter and other details of the iteration are available lexicographically before the domain of the corresponding 'DO'. This simplifies the problem of analysis considerably as, then, the nesting of 'DO's will not require a stack. Further the transfer instruction that was generated before the code for the data-list in the case of the GET statement is no longer necessary. It is necessary in I/O statementsbecause the domain of 'DO' group precedes the 'DO' specification.

In the case of the 'DO' statement, a short prologue and an epilogue are necessary. The prologue is responsible for updating the present DO counter. The DO counter is necessary for guarding against illegal GOTO's. After generating the prologue, the 'DO' analyses merges with the 'DO' coding for 'DO' group inside an I/O statement as the remaining coding is identical. Finally, after completing the code for the domain of the 'DO' the executive of second pass generates the epilogue.

The epilogue takes the form,

```
        TRA   CXXXXX
QXXXXX  EQU   *
```

where XXXXX is the 'DO' serial number of the 'DO' whose END statement was analysed by the second pass executive. The transfer restruction is necessary because the domain of the 'DO' is coded in the form of a closed subroutine. The code below illustrates this idea,

```
                [ Prologue ]

                [ Coding of DO consisting ]
                [ of calls to CXXXXX      ]

        CXXXXX  TRA   ***

                [ Domain of 'DO' ]

                TRA   CXXXXX
        QXXXXX  EQU   **
```

3.4 <u>Data Directed Input:</u>

As has already been mentioned the data directed input may not have any 'DO' group within its data list. If this were to be allowed, it gives to an ambiguity which is best illustrated by an example in which 'DO' group is specified in a data directed input statement. Consider,

```
GET DATA ((A(I),B(I) DO I = 1 TO 10))
```

Let A and B be arrays. Now corresponding to each data directed input command, a data set is available in the input stream which is terminated by the '$' character. When the run time routine R.DATA, which executes the command, encounters the '$', it returns control to the object code. It is implied that, on encountering a '$' in the input stream all input activities initiated by the GET DATA command will cease.

Under the circumstances the execution of the 'DO' group is controlled, not by the index of the 'DO' but by the input stream contents. In other words, even when the iteration count of a 'DO' is not exhausted a '$' in the input stream will terminate the entire input command. If we still persist with the 'DO' iteration count being fully exhausted, it will mean the reading of a data set in the stream which does not correspond to the input command currently under execution. Thus it is not meaningful to specify a 'DO' group within a data directed input command.

Secondly a subscripted scalar variable (i.e. an element of array) cannot occur in the data list. This is because the input stream itself will contain not only the name of an array but also its subscripts and the run time routine will store values only in those elements that are specified by the input stream. Under the circumstances it is illogical to specify subscripts in a data directed input. However, cross-sections of arrays may be specified in the data list, provided the input stream specifies only those subscripts which are varying in the data list. For example, for the command GET DATA(A(I,J,*)...); one would expect the input stream to be something like A(1)=..., A(2)=..., A(3)=..., ... We notice that although A is a 3-dimensional array the input stream behaves as though A were a one dimensional variable. This is because a one dimensional cross section has been specified in the data list.

<u>Description of the Data Directed Routine</u>:

The first pass processor, when it recognises a data directed I/O statement makes an exception to the rule that names of identifiers will not be transmitted to the second pass, and supplies the actual characters forming the elements in the data list. A structure (major or minor) cannot be used in a data-directed I/O command for the following reason. The first pass processor does not know that the variable it is handling is a structure and hence does not transmit the names of the base elements. When, in the second pass the GET DATA analyser recognises the structure it is too late for it to retrieve the names of the base elements. It can only access their attributes via the symbol table. Thus, it is logically impossible, in the present organisation to allow structure as data elements in the data list of a data directed I/O command.

For every data element in the data list a call is made to the name table search routine NAMSRH. The search routine maintains a table of names which have appeared in the GET DATA statements. The format of this table known as the Compile Time Data Table, is shown in Chapter II under the I/O section. This table contains all the relevant information regarding a variable like type, addressing scheme, and dimension information. If the data element transmitted by the analyser routine matches with any of the entries in Compile Time Data Table (CTDT) then the address of the corresponding entry in the Run Time Data Table (RTDT) is returned. For a match to take place, the attributes, the dimension information, the word count and the character count of the element in hand must match those in the particular entry in the CTDT. Note that the CTDT need not store the actual characters forming the variable. This is because names of two variables may match but their attributes and address in first order storage may not. However, if the attributes and addresses in first order storage do not match, there is no guarantee that the names also will not match. This is a consequence of the block structure of PL 7044. Thus name-matching is never conclusive and is, therefore not restorted to by the search routine. On the other hand, RTDT should have the characters of the data element as well as other information pertaining to the element. This is because during run time the input will identify the element only by name and the matching will have to take place on the basis of names only.

If, after searching the CTDT, the search routine fails to find a match, then an entry is made both in the CTDT and in the RTDT. The entry created in RTDT is passed on to the 3rd pass which then generates the code corresponding to that entry. Each entry in the RTDT has two label (in caseof an array), one for the first location of the entry and one for the start of the dimension information. For a scalar variable, only the first label is generated.

It should be noted that while the RTDT is a variable length table and each entry in RTDT has a variable length of words, the CTDT has a rigid format of 9 words and the total space available to it equals 500 words. If, the number of words required exceeds 500, an error condition will be raised and further processing of the statement will be suspended. The variable number of words in the RTDT is because of the fact that the characters comprising a name will vary from name to name.

During the 3rd pass, the code generating routine collects the Run Time Data Table entry addresses of the elements in a data list passed on by the analyser of the second pass and generates the following code,

```
        TSX  R.DATA,4
        PZE  n
        PFX  M1,,N1
         .
         .
         .
        PFX  Mn,,Nn
```

where,

n	=	the number of data elements,
M_i	=	the address of the Run Time Table Entry,
N_i	=	the address of the dimension information in the entry M_i,
PFX	=	PZE for scalar elements,
	=	MZE for array elements.

The 3rd pass is also responsible for the creation of the Run Time Data Table from the information passed on by the second pass analyser from time to time. The entries in the Run Time Data Table are all generated under a single location counter 'NAME'.

3.5 Edit Directed Input:

In this case the analysis is very similar to the list directed input although the semantics is somewhat different. Also a prologue has to be generated in this case which is absent in the case of list directed input statement. The prologue becomes necessary for the following reason. Every edit directed command has an associated format list which is either given along with input statement or specified remotely. In either case, a linkage has to be established between the data list and the format list before execution of the code corresponding to the data list can begin. But the nature of the format (immediate or remote) cannot be known until the data list has been scanned. (We always scan the input text from left to right once and the format specification is always preceded by the data-list). Hence, at run time, control must initially be transferred to that part of the code which performs the linkage. Thus, for example,

```
GET EDIT (A,B,C) (< format list>);
```

would be coded as follows:

```
        TSX   .IOSUP,4
        PZE   S.FBIN,,S.FSBL
        TRA   M00001          Transfer control to prologue
 M00002 EQU   *
        code for data  list
        TRA   M00003
 M00001 TSX   STHIO.,4
        PZE   <format list address>
        TRA   M00002
 M00003 EQU   *
```

Thus the analyser creates 3 additional labels -in the above example M00001, M00002, and M00003 are the three labels - in order to achieve the necessary linkage. Appendix M1 gives ~~th~~ the complete coding of an edit directed output command.

The remaining part of the analyser for the edit directed input is similar to that of list directed input that we discussed earlier. The code generated for the data list during the third pass, is however different from that generated for a list directed input, In particular, for a normal scalar variable the following code will be generated,

```
        AXT   BS.XX+YYYYYY,1
        PXA   ,1
        TSL   HNLIO.
```

BS.XX+YYYYYY is the first order storage allotted to such a variable.

For a formal parameter the code generated is,

```
        CLA   BS.XX+YYYYYY
        TSL   HNLIO.
```

Note that, in both the above cases the address of the data element is what is transmitted through the accumulator to the run time routine HNLIO. In this chapter on run time routines we shall learn more about this routine.

In case of arrays and array elements also, code is generated to place the address(es) of the element(s) in the accumulator before calling HNLIO. An example of the coding of a full array in an edit directed output command is give in Appendix M1. The variable W in that example is a 3 dimensional array.

Flow charts for the list, edit and data directed input command are given in Figs.3,4 & 5

4. Analysis of PUT Statement:

The PUT statement analyser has an organisation which is similar to the GET statement analyser. It has a common part for analysing the file usage, and the occurrence of a SKIP/LINE command. It has also three sub parts for analysing the three types of output commands namely the list-, the data- and the edit-directed commands. In the case of the data directed output, the analysis code actually merges with the analysis code of the input statement because the definition of data elements is the same for both input and output commands.

In regard to the list and edit directed output commands, some extra work has to be done by the output analyser vis-a-vis the input analyser. This is because expressions are allowed as elements of the data list in output commands. The method by which the expressions in a PUT LIST or PUT EDIT are handled is best explained with the aid of the following example:

```
PUT EDIT ( ( ( A+B ) * C )/D, ... )(f-1);
         1 2 3     3'    2'        1'
```

The analyser first consumes the 3 left parentheses. Now '$(_{1}$' is always required for enclosing the data list. Thus we are left with two excess left parenthesis. These may correspond to either two 'DO' groups or to an expression involving two pairs of parentheses or one 'DO' group and an expression involving one pair of parentheses. The recognition problem is solved by calling H.PUT which is an entry point in the buffer handler. H.PUT returns at $)_{3'}$ because it is unable to proceed further in the absence of a matching left parenthesis. Also EX.PR flag is turned on by this routine. The PUT routine on sensing that EX.PRK is on, supplies the left parenthesis, reduces the count of excess left parenthesis by one and calls back the buffer handler at H.PUTN in order to ascertain if the expression continues beyond the right parenthesis 3'. H.PUTN returns at $)_{2'}$ for the same reason that H.PUT returned. The action taken by the PUT analyser is also the same. Finally H.PUTN returns after encountering a comma. At this point the analyser calls the arithmetic processor which generates code for the expression and returns with the address of the temporary word where the result of the expression is made available at execution time. This temporary word is then used by the analyser to generate the code to initiate calls to run time routines just as in the case of scalar variables. Thus, the code for the above expression is as follows:

```
[arithmetic code]

AXT  TS.XX+YYYYYY,1

PXA  ,1

TSL  HNLIO.
```

where TS.XX+YYYYYY is the address of the temporary word in which the result of the expression is available.

Array expressions are handled in a fashion similar to scalar expression. The only difference comes about in the additional code generated for the repetition involved in array expression. In this case, the arithmetic processor returns a label in which the address of the result of the expression is stored. Thus the code generated will be,

```
[arithmetic code]

 LXXXXX  AXT  **,1

         PXA  ,1

         TSL  HNLIO.

[repetitive code
 generated by
 arithmetic
 processor.    ]
```

Finally, we come to the case of the structure expression. In this case H.PUT or H.PUTN turns on both the expression flag

EX.PR and STRFLG. In this case the arithmetic processor is called repeatedly with different sets of base elements of the structures involved and codes are generated for each base element as if they were ordinary variables.

Except for the handling of expression used as data elements, the PUT EDIT/LIST analyser is similar to the GET analyser. In fact the coding for 'DO' group in a PUT EDIT/LIST command is merged with that for the input routine. Other codings that are shared by the two analysers are the STACK and UNSTCK routines discussed earlier and the various pointers and counters discussed under NOTATIONS in Sec. 3.

5. READ and WRITE Routines:

These routines handle record input and output statements. The syntax of these statements is straightforward when compared with the syntax of GET/PUT statements. File checking is carried out as in the case of GET/PUT statement, at the time of file declaration and file status block generation only. Code is then generated to initiate a call to the run time routines for reading in or writing from the array mentioned in the command, a single record of information. The run time routine used is the Fortran binary read/write (RWB) routine with slight modification.

6. Error Recovery:

A number of error conditions are recognised by the routines described in the last three sections. The action taken in each case is to give a relevant error message, and skip the rest of the statement. The various error messages that are given are listed below:

1) Illegal use of Operator: This occurs when the analyser expects an operand and an operator occurs or when the analyser expects an operator of one kind ana encounters a different operator.

2) Unidentifiable Operand: This occurs when an expected key word is missing or misspelled or some other key word is used in its place.

3) Illegal use of operand: This occurs when the analyser expects an operand and hits upon an operator.

4) Too many files in an I/O Statement: More than one file name is used in the same I/O statement.

5) Both SKIP and LINE commands may not be used. This is self-explanatory.

6) Conflicting usage and declaration of file: This occurs when the analyser checks on the usage of the file before generating the file card.

7) Expression in GET statement: Expressions are not allowed in GET statement.

8) Parentheses Mismatch: This occurs when the parentheses are not balanced.

9) Literal used as Data Element: Literals cannot be used as data elements.

10) Illegal occurrence of 'DO'; This is probably due to a 'DO' group being used without enclosing pair of parenthesis.

11) Stack over-flow: This occurs when an attempt is made to nest more than 5 'DO' groups.

12) Literal used as 'DO' index: Self explanatory.

13) Attempt to input function name: Self explanatory.

14) Incorrect ending of I/O statement: This occurs when the semicolon is encountered before the closing right parenthesis.

15) Illegal type of 'DO' index: 'DO' index variable can only be either integer or floating point variable if the TO-BY option is to be used.

16) Dimensioning error in Array name in GET/PUT DATA: This occurs when attempt is made to specify a single element of an array as data element in a DATA directed I/O.

17) Array variable used as subscript: Self explanatory.

18) Illegal variable used as format label in edit directed I/O: The label associated with a format can only be a label variable.

Fig. IO-1

Common branch for GET LIST/DATA/ EDIT commands

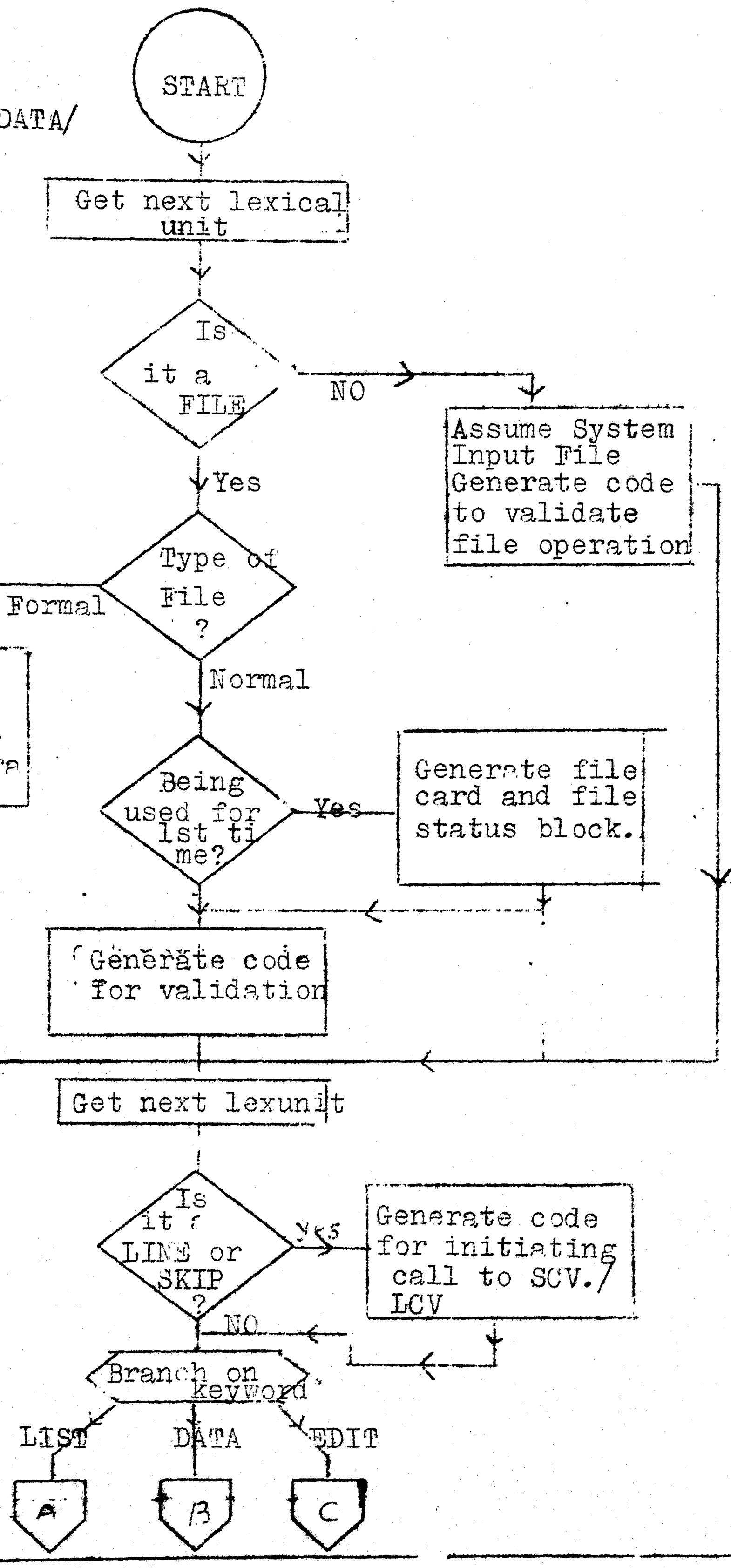

START
Get next lexical unit
Is it a FILE
NO
Assume System Input File Generate code to validate file operation
Yes
Type of File ?
Formal
Generate code access file word and validate file operation.
Normal
Being used for 1st time?
Yes
Generate file card and file status block.
Generate code for validation
Get next lexunit
Is it a LINE or SKIP ?
Yes
Generate code for initiating call to SCV./ LCV
NO
Branch on keyword
LIST
DATA
EDIT
A
B
C

Fig. IO-2 List-directed input

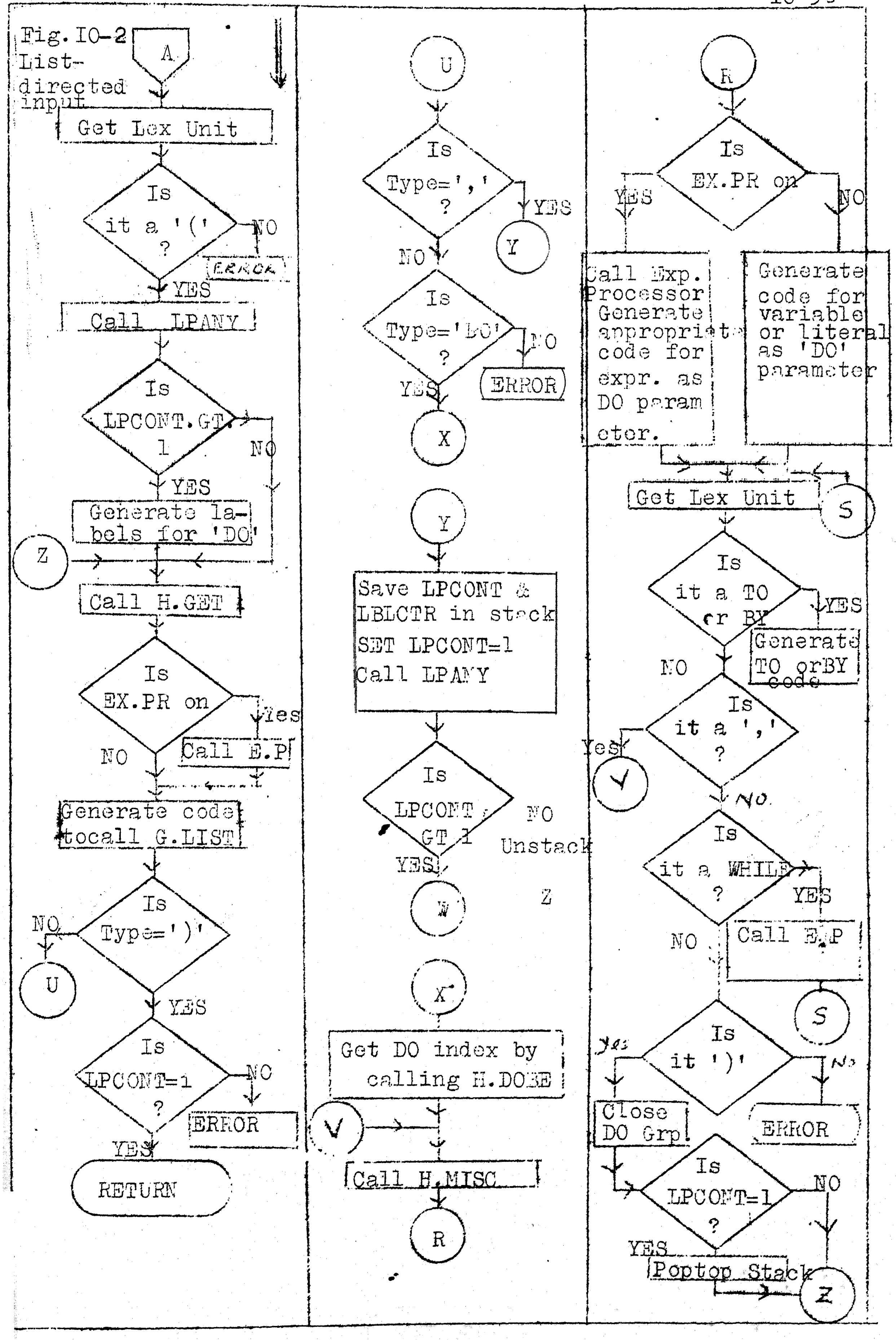
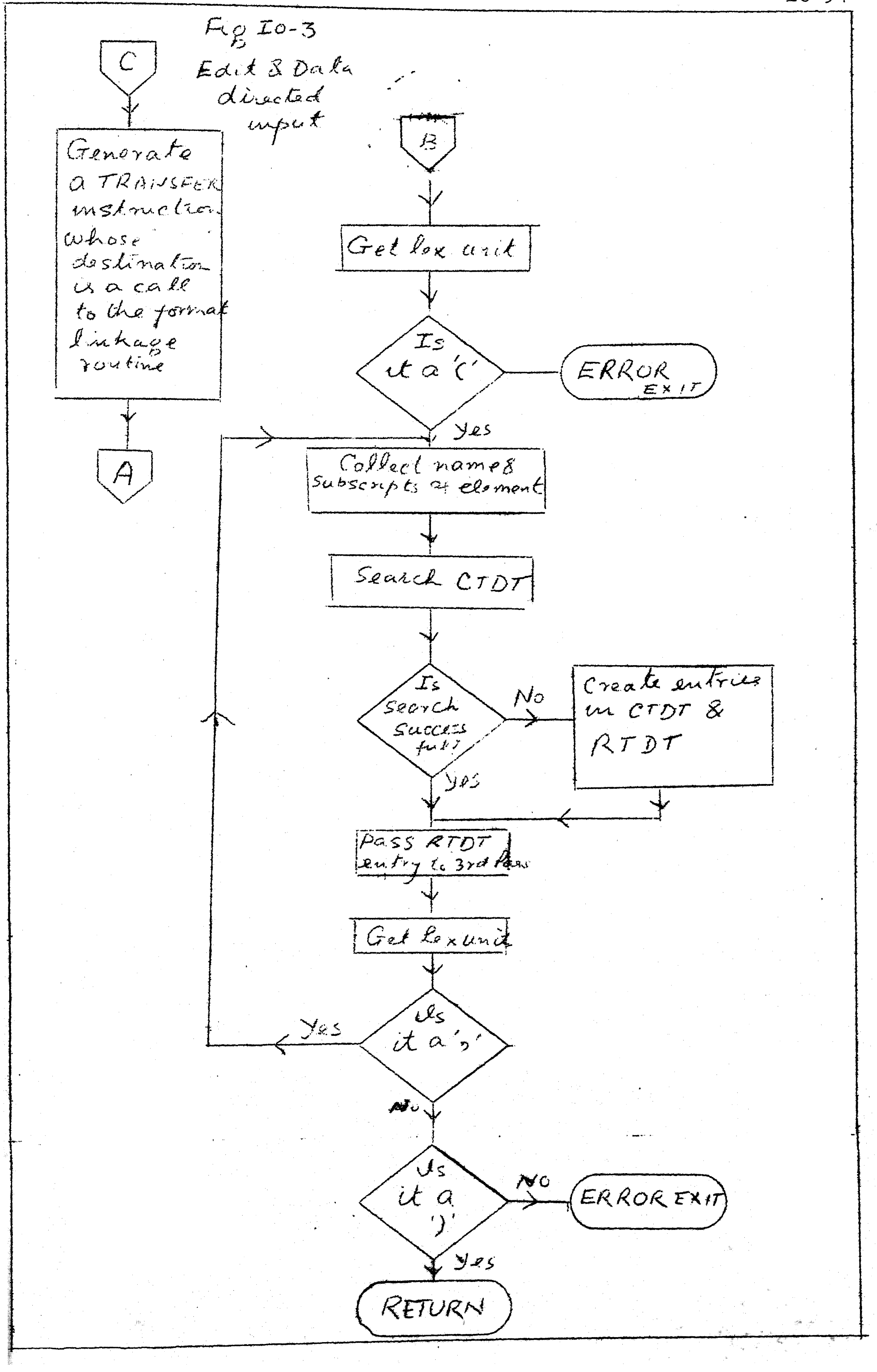


Fig IO-3

Edit & Data directed input


C
Generate a TRANSFER instruction whose destination is a call to the format linkage routine
A
B
Get lex unit
Is it a '('
ERROR EXIT
Yes
Collect name & subscripts of element
Search CTDT
Is search successful?
No
Create entries in CTDT & RTDT
Yes
Pass RTDT entry to 3rd Pass
Get lex unit
Is it a ','
Yes
No
Is it a ')'
No
ERROR EXIT
Yes
RETURN

CHAPTER V

THE FORMAT TRANSLATOR

1. Introduction:

PL 7044 provides a wide range of format items which can be used with the GET/PUT edit statements for complex editing of input and output data items. As in Fortran, the basic strategy is to translate the format items into a list, called the format list which is essentially a sequence of calls to various run-time conversion and editing routines. Unlike Fortran, however, PL 7044 format statements are not, in general, logically independent of the rest of the program and hence they cannot be translated into MAP code in one scan. In other words, whereas in Fortran, format statements cannot include variables and expressions as field count, group counts or explicitors, in PL 7044 one can have these types of specifications also. The reason for incorporating these features in PL 7044 is that there is no provision for run time or read-in formats as is the case in Fortran. But it turns out that the features in PL 7044 are really the more flexible of the two. This is evident from the fact that the variables used in a format statement can themselves be read in and in general manipulated with, as any other variable. This gives rise to a new dimension in format specification. Moreover, the run time formats of Fortran have, perforce, to be translated at execution time, thus slowing down the program as a whole, whereas this is not true in case of the pretranslated formats of PL 7044.

In general, then, the code outputted at the end of the analysis phase for a format statement, will be interspersed with the code generated by the expression processor and another scan is necessary for separating the two and generating the MAP code. This very factor precludes the simple one shot technique adopted in translating formats in Fortran.

In the following pages, first the syntax and semantics of format items are described briefly in relation to their Fortran counterparts , then the format analyser of PASS II is taken up along with a case study and finally the format list generator of PASS III is described.

2. Syntax and Semantics:

General format of a format statement is

[remote format specification/immediate format specification]

General format of a remote format specification is

FORMAT (format-list);

General format of an immediate format specification,

... (format list);

Meaning:

A remote format specification specifies that the items to be formatted are remote from the format list, whereas the immediate format specification implies that the [illegible] data items are those that just precede the specifications.

General format of a format list is,

Item [, item] or item [item]

where " item" may be

format item

n format item

n (format list)

where n represents an iteration factor which is either an expression enclosed in parentheses or a simple integer constant.

Meaning:

When n is used with a format item it is equivalent to the familiar Fortran field count while when n is used with a format list it is equivalent to the Fortran group count. A zero or negative field or group count specifies that the associated field or group is to be skipped. Note that in Fortra the rule is that the associated field or group will be used onc even if the count is zero. In the ensuing discussions, n will be referred to as either the field count or group count and not as an interaction factor.

Format items may be classified as data format items and control format items. Before describing data format items a few notations are in order.

The letter w represents the field length of the data ite in the input/output stream. This includes sign position, decimal/binary points, blanks and 'E' used in E format items.

The letter d represents the number of positionsafter the decimal point. On input, the data item in the data stream is treated as if it conformed to the characteristics described by the format item.

a) Integer Format Item:

General Format	Comment
I(w)	Same as in Fortran

b) Floating Point Format Item:

General Format	Comment
F(w,d)	Same as in Fortran

c) Floating Point with Exponent:

General Format	Comment
E(w,d)	Same as in Fortran

d) Complex Format Item:

General Format

C([Control Format List] ,E/F Format Item [,Control Format List] ,E/F Format [,Control Format List]).

<u>Meaning</u>:

This feature is not available in Fortran. The C Format field must have exactly 2 E or F Format items and can have control format items before, after and in between them.

e) Octal Format Item:

General Format	Comment
Ø(w)	Same as in Fortran

f) Character String Format Item:

General Format	Comment
A(w)	Comments reserved till Chapter on Run Time Routines (ACV Routine)

g) Bit String Format Item:

General Format	Comment
B(w)	This feature is not availabl in Fortran

The above format item is used when we want to output or input a bit string data item. If used with other data items unpredictable results will be obtained.(See Chapter on run time routines (BCV routine)).

Control Format Items:

These are non data transmission format items and are chiefly used for editing:

a) SKIP Format Item

General Format	Comment
SKIP(w)	It skips w on the output fil or skips w cards on input fi

b) LINE Format Item

General Format	Comment
LINE(w)	It skips to the w-th record on the output file or to w-th card on an input file.

c) COLUMN Format Item

General Format	Comment
COLUMN(w)	It positions the column pointer to the w-th character position of a record in an input file or writes the next data item at w-th character position of the line.

Note: In item (b) if w is less than the present record number, the LINE command will be ignored. In item(c) if the present character position is greater than w, then that record is skipped (if input) or written out (if output) and the pointer positioned to the first character of the next record.

d) X Format Item

General Format	Comment
X(w)	Same as in Fortran

e) PAGE Format Item

General Format	Comment
PAGE	Can be used only with output print files and signifies the ejection of a page.

General Comments:

In the above items the syntax and semantics do not conform entirely to the PL/I Language Specifications. They

They represent a compromise between the sophistication of PL/I and the primitivity of Fortran IV.

In the subsequent discussions, w, and d are called the first explicitor and second explicitor respectively.

3. The Format Analyser:

Figures FM-1 a thru 1.f show the flow chart of the format analyser of PASS II. In the following descriptions references to 'Lexicon' and to lexical units must be understood to be references to Second Pass Executive and the information supplied by it respectively. The Second Pass Executive simulates faithfully the functions of the lexicon of Pass I besides performing other vital tasks. The following notations are used in the description and the flow charts.

Notations:

FLAG :	flag to indicate that the field count, group count or explicitor is a literal.
VFLAG :	flag to indicate that the field count, group count or explicitor is a variable. (when both these flags are off, it indicates that the field count group count or explicitor is an expression).
CMFLAG:	comma flag to indicate the validity of an occurrence of a comma.
CFLAG1:	-ve if the first item of a C format item is yet to be specified.

CFLAG2:	-ve if the second item of C format field is yet to be specified.
FMTF.1:	-ve indicates that both fields of a C format item have occurred but the enclosing right parenthesis is yet to be scanned.
H.MISC:	external routine (in BUFFER HANDLER of the expression processor) to investigate nature of the group count, field count or explicitor.
RINR.A:	external routine to process expression and return the label number where the result of the expression will be stored in the address part.
RINR.D:	external routine to process expression and return the label number where the result of the expression will be stored in the decrement part.
FMTFLG:	flag to indicate remote or immediate format statement.
EX.PR :	expression flag turned on by H.MISC to indicate presence or absence of an expression.

Description of the Format Analyser:

As has been described in the 'Syntax and Semantics' section, a large number of format items with different semantics but almost the same syntax have to be processed. For example, the format items A,I,B,Ø,X etc. all have the same syntax but the code to be generated in each case is different. Also any of the format items can have expressions as explicitors. Thus one or two service routines which can be called from various points in the analysis, were incorporated to make the coding of the analyser compact and efficient. We shall, therefore, first describe the service routines before outlining the analysis procedure.

F.EXP: Routine to Initiate a Call to BUFFER HANDLER:

The routine initiates a call to H.MISC in order to determine the nature of the explicitor, group or field count. H.MISC processes the input from the 'code' file prepared by the First Pass Processor and lays out the lexical units in a format recognisable by the arithmetic processor. It also puts on a flag EX.PR if an operator is recognised in the course of its processing. An array element used as an explicitor also causes this flag to be turned on by H.MISC. When H.MISC returns control to F.EXP this routine examines whether EX.PR is on or off. If it is on, then it writes certain end markers in the buffer prepared by H.MISC and initiates a call to the actual arithmetic processor RINR.A or RINR.D as the case may be. This call and the preparation for it are made through two other service routines described later in this section. The entry points RINR.A and RINR.D in the arithmetic processor are responsible for generating codes for calculating the expression, converting the result into integer, if necessary, and if the integer is greater than zero, to store the integer in the address or decrement portion of a labelled location. The label number is returned by these two routines. If EX.PR is not on, then F.EXP assumes that the explicitor must be a literal integer or a variable. It checks the appropriate locations in the buffer prepared by H.MISC for the type of operand processed by it. If the TYPE is 1 or 2 it signifies that a variable was involved. It then picks up the address of the symbol table entry, turns VFLAG on, FLAG off and returns. If TYPE is 3 or 4 a literal constant is indicated. If type is 3, then the literal is an integer constant and if it is 4 then a floating point constant is indicated. In the latter case, the constant is converted to an integer and if it is greater than zero, then it is saved in a preassigned location, FLAG is turned on, VFLAG turned off and control is returned to the caller. If, however, the literal is less than zero, then the preassigned location is zeroed and an exit from the routine is taken.

A.ARIT: Service Routine to Initiate Call to Arithmetic Processor:

In the previous paragraph, it was mentioned that if EX.PR is on, then a call to arithmetic processor is initiated. This is done through this routine. Before actually calling the arithmetic processor, this routine writes out end-markers at the end of the buffer containing information processed by H.MISC, updates the pointers to the beginning and end of the buffer. The result returned by the arithmetic routine is made available in a preassigned location for further processing. This routine then generates code to indicate to Pass 3 executive that control must be transferred back to the format code generator of Pass 3.

This is because the intervening arithmetic code would have preempted control from the format generator. This routine then transfers control back to F.EXP.

D.ARIT:

This routine is identical to A.ARIT except that a call is initiated to RINR.D instead of to RINR.A. The result of the arithmetic coding will now be stored in the decrement part of the run time labelled location instead of the address part.

The choice between A.ARIT and D.ARIT is indicated by a switch which is set by the caller to F.EXP. With the help of these 3 service routines the analyser will function as follows.

Two entry points are provided one for the remote format specification and the other for the immediate format specification. The need for two distinct entry points arises because in the case of the immediate format specification, the opening left parenthesis has already been consumed by the calling routine (GET/PUT) whereas in the former case this is not so. Secondly, for a remote format statement, a transfer instruction to bypass the format code has been generated by the executive routine of Pass II and the destination label corresponding to this transfer instruction must be generated by the format analyser at the end of the format code. In the case of the immediate format specification, this function is carried out by the calling routine itself.

After taking note of the above distinction by setting FMTFLG off or on, the two entry points merge together. The next lexical unit is examined for an operator or an operand.

Analysis of Operands:

When an operand is indicated it could only be a keyword corresponding to the formats A,I,Ø,B etc. or an integer or a hollerieth field with character string or a hollerieth field with bit string. Let us first examine the path taken by the key word operands.

A,B,I,Ø Format Items:

These require a single explicitor enclosed in parenthesês. Further they cannot occur within a complex format field. Therefore, before looking for the explicitor, the validity of their occurrence is checked by testing the flags CFLAG1, CFLAG2, FMT.F. All three must be in the off condition. If this test fails an error condition is raised (see section on 'error recovery'). In order to determine the nature of the explicitor, a call to the service routine F.EXP is made. Upon return from F.EXP, a test for the presence of the closing right parenthesis is made. If this test is successful, and if FLAG is on,code is generated to indicate the occurrence of the format item with literal constant as the explicitor. If FLAG is off and VFLAG is on, code to indicate format item with variable explicitor is generated. This is followed by the symbol table information regarding its addressing mode (direct or indirect), first order storage area and type of variable. If both flags are off, then a code to indicate format item, with expression as explicitor, is outputted followed by the label information where the expression processor will store the result of the expression. Thus each of the format items is allotted 3 distinct codes corresponding to 3 types of explicitors.

X, LINE, COLUMN, SKIP Format Items:

These are similar to the foregoing format items except that since they do not operate upon data items, they can occur even inside a C-format field. Hence the initial part of testing whether a C-field is active or not is unnecessary here.

In all the above format items, there is a striking similarity in the task to be performed. Hence a macro was introduced to handle all the format items. The arguments of the macro were 3 unique numbers corresponding to the 3 types of explicitors. Thus for example, A format was given numbers 7,8 and 9. Similarly other format items were given numbers in ascending order so that the format generator of 3rd pass could easily decode the output of Pass 2 with the help of a table.

E-and F-Format Items:

These format items require a little more coding than the others because these have two explicitors and thus give rise to 9 possibilities. These are as follows:

1) First explicitor constant, second explicitor constant,
2) First explicitor variable, second explicitor variable,
3) First explicitor expression, second explicitor expression,
4) First explicitor constant, second explicitor variable,
5) First explicitor variable, second explicitor constant,
6) First explicitor expression, second explicitor variable,
7) First explicitor constant, second explicitor expression,
8) First explicitor variable, second explicitor expression,
9) First explicitor expression, second explicitor constant.

Thus, it is clear that nine different opcodes are necessary for each of the nine different situations.

Before actually processing the explicitors, it has to ascertain whether the C-format item is active. This is done by successively testing CFLAG1, CFLAG2 and FMTF.1. If all the 3 flags are off then C-format field is not active. If CFLAG1 is on, the routine turns it off and proceeds. If CFLAG2 is on then the routine turns it off and proceeds. If both are off and if FMTF.1 is on, an error condition is raised, since it signifies more than two E or F format items are present, in a C-field which is against the syntax rules.

After the above tests are over, the presence of an opening left parenthesis is confirmed. Then a call to the service routine F.EXP is made. On the basis of the flags set up by F.EXP the partial code corresponding to the first explicitor is formed and stored in the output area. A second call to F.EXP is made to ascertain the nature of the second explicitor. The information so gathered is again suitably coded and stored in the output area. Finally depending on the combination of the types of the 1st and 2nd explicitors a unique code is outputted and is followed by the information gathered during the two calls to F.EXP.

When Operand is an Integer Constant:

The occurrence of an integer outside the parenthesis delimiting an explicitor can only signify the occurrence of a group or field count. In order to resolve this uncertainty, the next lexical unit is examined. If it is a '(' then it signifies a group count and code is generated accordingly.If it is anything other than a '(' then it is a field count and code is generated accordingly.

When Operand is a Character Constant:

When an operand is a character constant, code is generated to indicate occurrence of a hollerieth character field. The actual character constant and the number of characters are also passed on to Pass 3.

When Operand is a Bit String Constant;

When an operand is a bit string constant, code is generated to indicate a hollerieth bit string field. The actual bit string and the number of bits are also outputted.

Analysis of Delimiters:

Three valid delimiters can occur inside the format statement (other than those that can occur in an expression used as an explicitor, field count or group count which is handled by H.MISC). The three delimiters and the action taken are as follows:

When the Delimiter is a Comma:

The comma is used to separate two format items. The presence of a comma is optional. In no other situations can the comma occur in a format statement. The valid occurrence of a comma is indicated by CMFLAG. As soon as a comma is detected the flag is turned on. After every format item the flag is turned off. No code is generated on detection of a valid occurrence of a comma.

When the Delimiter is a Right Parenthesis:

The right parenthesis has 3 functions. These are a) to close an explicitor b) to close a group of format items, c) to close a C-field. The first function does not come under our purview, here, since the closing parenthesis of an explicitor is always consumed by the expression handling routine. Thus, when a right parenthesis occurs in the format analysis, it must signify end of a C-field or end of a group. Now a C-field can be closed only if both its sub-items have occurred. This situation is indicated by CFLAG1 and CFLAG2 being off and

FMTF.1 being on. If this is the situation, then the action taken is to put off FMTF.1 and return for further processing. If on the other hand CFLAG1 or CFLAG2 or both are on then an error condition is raised. If all the three flags are off, then it can only mean end of a group count or end of the format statement itself. The parenthesis count is reduced by one and if it becomes equal to zero, an end of format statement is indicated. If the count is greater than zero an end of group is indicated. In either case the appropriate code is generated.

When the end of a remote format specification is sensed, code to generate the label given by the executive of Pass 2 is also outputted whereas in the case of the immediate format specification a simple return to the caller is made.

When the Delimiter is a Left Parenthesis:

The occurrence of a left parenthesis in a format statement can signify 3 situations a) opening the description of an explicitor b) opening a group c) opening the description of a field count or a group count.

The first situation is handled always by the corresponding format item routines described earlier. The second and third situations are always handled in relation to one another. We initially assume that whenever a left parenthesis is recognised at the main branch point Fig. FM-1a it must refer to the third situation. Under this assumption, a call to F.EXP is made to find the nature of the field or group count. This will return with information about field count or group count. The last lexical unit met by F.EXP will be a right parenthesis. Now, in order to resolve the apparent ambiguity between field and group count the next lexical unit is examined. If it turns out to be a left parenthesis, then it is clear that it signifies the opening of a group and therefore the description obtained by F.EXP pertains to a group count. Here we see how the situation (b) in the above classification has been taken care of. If the next lexical unit is not a left parenthesis, then it can only be a field count description. Thus (b) and (c) always occur together and never in isolation.

We have now dealt with all the situations possible during the format analysis. We shall now briefly enumerate the error conditions that may arise during analysis and the actions taken.

4. Error Recovery:

a) Illegal Operator/Delimiter in Format Statement:

This error occurs when an operator/delimiter other than '(', ')' or ',' is encountered while analyzing a format statement. An error message is given and processing of the statement is terminated.

b) Illegal Operand in Format Statement:

An operand other than those enumerated in the last section results in this error. An error message is given, and further processing is terminated.

c) Bit or character string not allowed in field or group count. The action taken is the same as in the previous two cases.

d) Arrays or structures not allowed in Format Explicitors. Action taken is the same as above.

e) Illegal format item within C-field. Only E or F or X, LINE or COLUMN format items are allowed within C field.

f) Misplaced Comma: This occurs due to invalid use of a comma. The comma is ignored.

g) Picture formats not allowed: In the present version no provision for handling picture items have been made.

h) Parenthesis Mismatch: The number of left parentheses does not equal the number of right parenthes.s.Processing is terminated.

i) Too few sub-fields in a C-format: There should be exactly 2 E or F fields inside a C-format item.

j) Too many sub-fields in C-format: Attempt to have more than 2 E or F fields in a C-field results in this error. Processing of the statement is terminated after putting out a message on the output file.

5. <u>Case Study</u>:

In this section we shall study the second pass output generated for one format statement having all the features discussed under section 3.

Consider the format statement:

(5((M)(X(10),COLUMN(M),C(F(10,M),E(M,<exp>)),

'RESULT', A (<exp>), B(3), 5 ∅(12)))),.

The code generated and the comments for each item are given below:

Item No.	Code in Octal	Comments
1	777777777777 0 00000 0 00022	Marker word followed by code to indicate that format statement begins.
2	0 00005 0 00004	Address part indicates integer group count, decrement part contains the actual group count.

3	0 00000 0 00005
	0 00144 0 00102
	0 00000 0 00001
		Address part indicates variable group. Count, two word description of variable. First word, 5th character is block no., 4th character is address bit, decrement contains offset. Second word contains minor type.

4	0 00012 0000020	Address part indicates X Format with integer explicitor, decrement part contains the integer.

5	0 00000 0 00026
	0 00144 0 00102
	0 00000 0 00001
		Address part indicates columnitem with variable explicitor. Next two words are variable descriptors (See item 3).

6	0 00000 0 00037
	0 00000 0 00012
	0 00000 0 00000
	0 00144 0 00102
	0 00000 0 00001
		Address part indicates F format item with first explicitor an integer, second explicitor a variable. First word is the integer, second word is dummy. Two word description of variable M (See item 3).

	777777777777 (Arithmetic code for calculating expression).	

Item No.	Code in Octal	Comments
7	777777777777 0 00000 0 00022	Marker word followed by indicator.
	0 00000 0 00054	Indicates E Format with first explicitor variable, second an expression.
	0 00144 0 00102 0 00000 0 00001	Descriptor of variable (See Item 3).
	0 00000 0 00001 0 00000 0 00000	First word is label number, second word is dummy
8	0 00000 0 00002	Indicates field count with variable field count.
	0 00144 0 00102 0 00000 0 00001	Two word description of variable (See Item 3).
9	0 00000 0 00013	Indicates I-Format item with variable explicitor.
	0 00144 0 00102 0 00000 0 00001	Two word description of variables.
10	0 00006 0 00064	Indicates H-Format item with count of Character in decrement.
	512562644363	Octal representation of 'RESULT'.
	777777777777	
	(Arithmetic Code)	
11	777777777777 0 00000 0 00022	To indicate control to be transferred back to format.
	0 00000 0 00011 0 00000 0 00002	Indicates A format with expression, label number associated with expression.
12	0 00003 0 00015	Indicates B Format, with decrement containing integer.
13	0 00005 0 00001	Indicates integer field count with decrement containing integer.
14	0 00012 0 00057	Indicates Ø format with decrement containing the integer.
15	0 00000 0 00063	End of a group.
16	0 00000 0 00063	End of another group.
17	0 00000 0 00062	End of a format.

6. The Format List Generator:

The task of the Format List Generator is greatly simplified by the format analyser. The generator routine which is a part of the 3rd pass, has to produce the final MAP coding corresponding to the input supplied in the form shown in the preceding section. The generator recognises 53 different opcodes, produced by the format analyser. This is done with the help of a table of 53 entries, each entry responsible for producing the desired code depending on the information contained in the subsequent words in the input stream. Since it is neither feasible nor necessary to explain each of the 53 entries individually, we shall restrict our attention to a few representative examples taken from the Case Study of the previous section.

Example 1:

Consider item 7 in the Case Study of Section 5. The marker of all 1's followed by thw word containing $(22)_8$ directs the 3rd pass executive to transfer control to the format generating routine. The first word read by this routine has $(54)_8$. This indicates an E conversion whose first argument is a variable and the second is an expression. The next two words following this word has a description of the variable in packed form. And the next word contains a label number which is to be concatenated with L and used to label the argument location of the calling sequence of the E-conversion routine. Thus the code generated will be,

```
              AXT  0,1           initialize XR1
              MIT* BS.02+000100  is the variable > 0
              LXA  BS.02+000100,1 yes load XR1
              SXA  *+2,1         save in argument location
              TSX  IOHEC., 4     call E-conversion routine
       L00001 PZE  **            argument location with label
              EXTERN IOHEC.
```

In the above example, since the minor type specified that, it was an integer variable no coding for conversion was necessary. Also since the indirection bit was 1, the address BS.02+000100 is to be used indirectly.

If the variable had been a floating point variable with indirection bit off, then the following code would have been generated,

```
       CLA  BS.02+000100   Bring variable
       UFA  =Ø233000000000 Convert
       ALS  10
       ARS  10
       AXT  0,1
       TMI  *+2            test
       PAX  ,1             load XR1
       SXA  *+2,1          fill argument
       TSX  IOHEC.,4
L00001 PZE  **
```

Here a bit of unoptimised code is discernable when we load XR1 from the accumulator and then store XR1 into the argument. But this has been done for the sake of uniformity in the decrement field.

The code generated will be

```
TSX  IOHBCO.,4
PZE  3
EXTERN IOHBC.
```

<u>Example 4</u>:

Consider item 4 having a H-Format.

The first word, besides indicating the occurrence of a hollerieth field, also gives the number of characters involved. From this it is possible to find the number of words w also. The routine then reads in the next w words from the input file and generates the following code,

```
TSX  IOHHC.,4
PZE  6
BCI  1,RESULT        w=1 here
EXTERN IOHHC.
```

<u>Example 5</u>:

Consider item 3. The first word indicates a variable group count. The next two words describe the variable. The code produced is,

```
AXT  0,1
CLA* BS.02+000100
TMI  *+2
PAX  ,1
TSL  IOHLP.
EXTERN IOHLP.
```

Example 6:

Consider item 8 which indicates a field count with a variable count. The code generated is as follows:

```
AXT  0,2
CLA* BS.02+000100
TMI  *+2
PAX  ,2
```

Example 7:

Consider item 16 and 17.

The code generated for item 16 is just one instruction,

```
TSL  IOHRP.
```

while for the end of format it is

```
TRA  IOHEF.
```

For a description the run time routines referred to in the above example, the reader is directed to the Chapter on Run Time Routines. A Complete translation for the Case Study example appears in Appendix N.

Finally let us briefly look into the organisation of the format generating routine of Pass 3.

A number of service routines form the nucleus of the format generator and these routines are called from the various points in the program. The routines which fall in this category are (a) an address mechanism sorting routine (b) a conversion routine ~~x~~(c) an expression handling routine (d) special routines for E- and F- formats.

a) This routine determines whether the indirection bit is on or off and accordingly generates appropriate code.

b) This routine handles the necessary conversion to integer type. The codes generated by this routine include floating to integer conversion, character to integer conversion and bit string to integer conversion codes.

c) This routine handles codes pertaining to generation of labelled locations as directed by the expression processor.

d) These routines handle the four words which always follow the code indicating E- and F-format items. The first two words pertain to the first explicitor and the next two to the second explicitor. Depending on which, of the nine combinations, is specified, these four words are interpreted accordingly and code generated.

Normally error conditions cannot arise unless there is an error in tape transmission. Such errors are sometimes detectable and sometimes not. As and when they are detected, the job is dumped after outputting a suitable message.

Fig FM-1a
Format Analysis

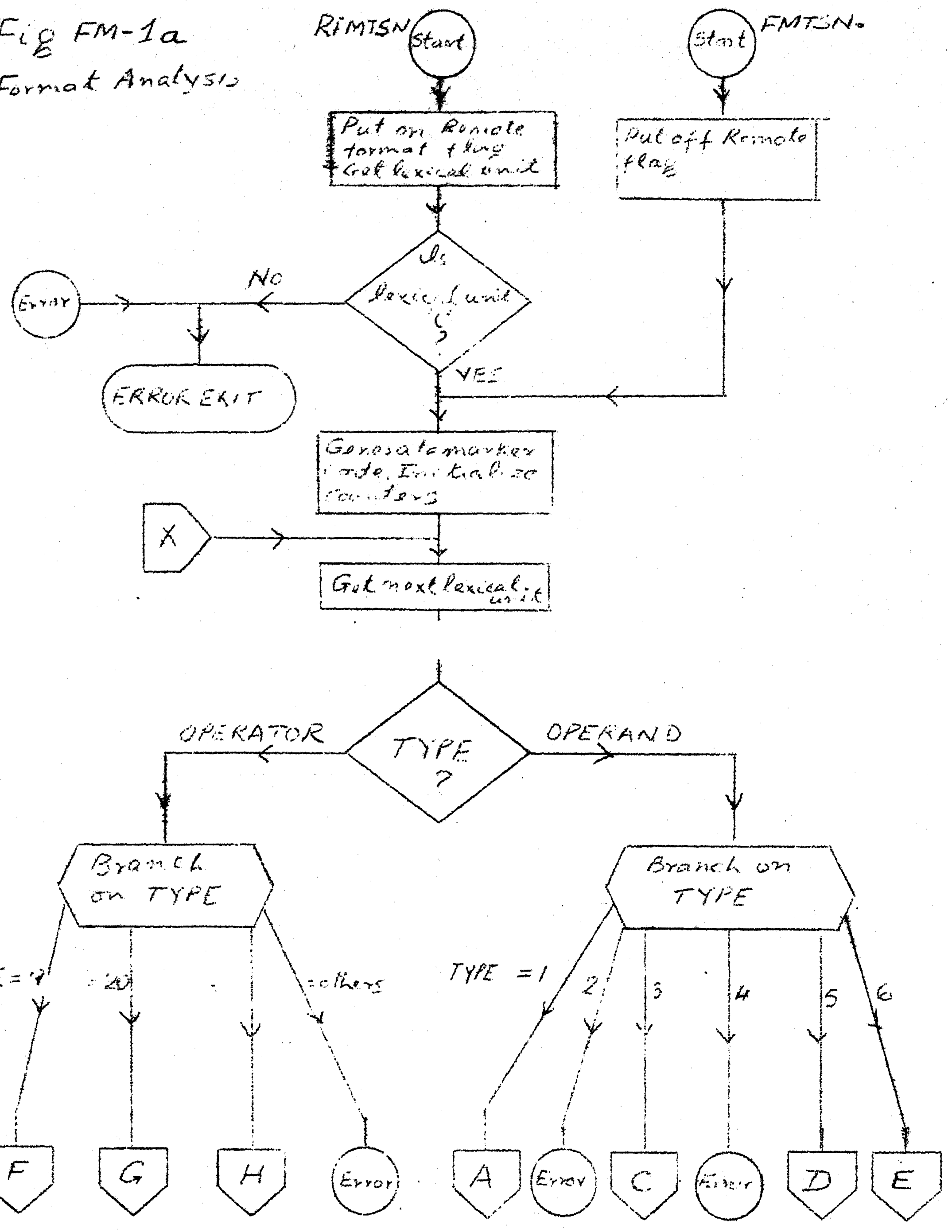


Fig FM-16
FORMAT ANALYSIS

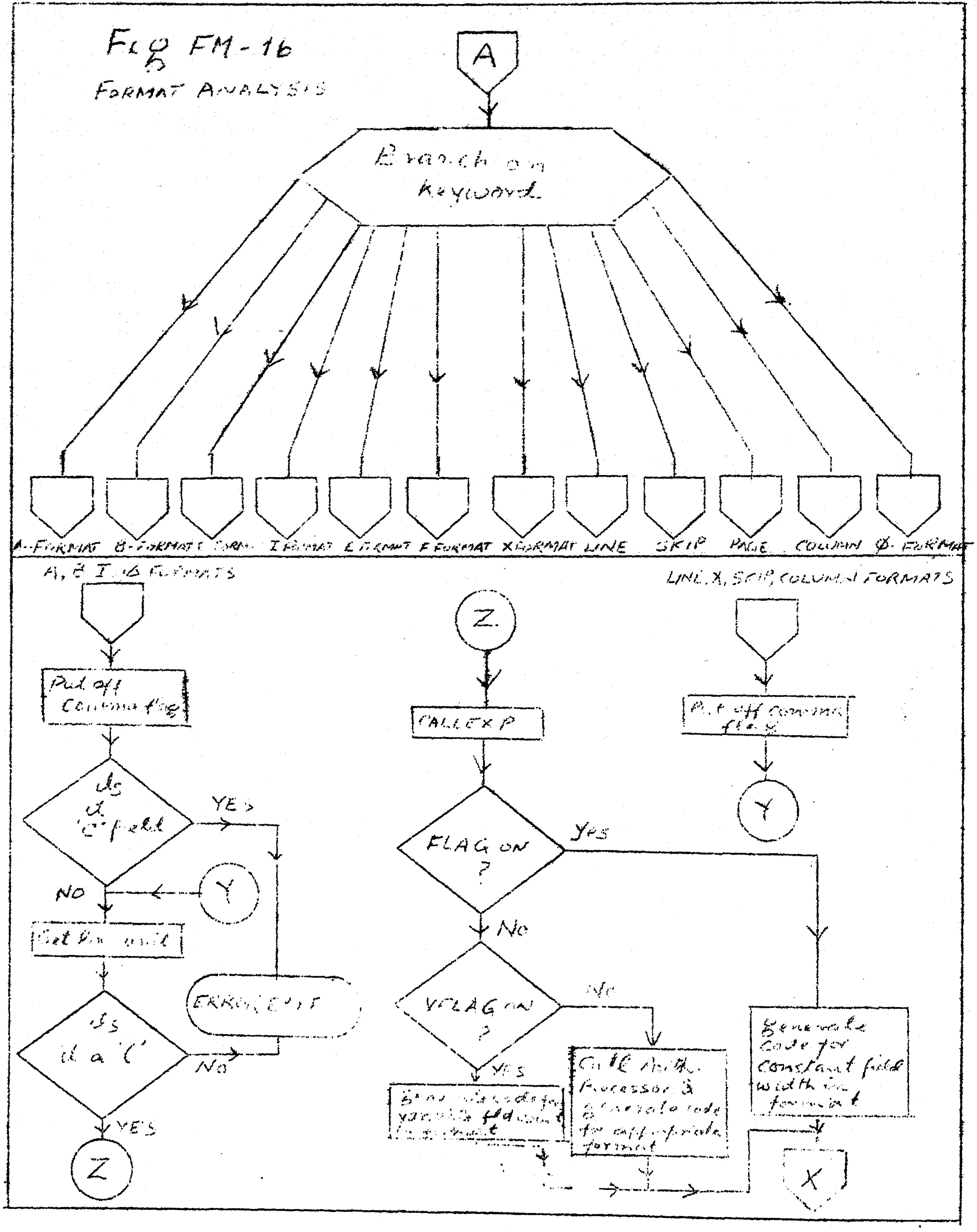


Fig FM-1C
FORMAT ANALYSIS

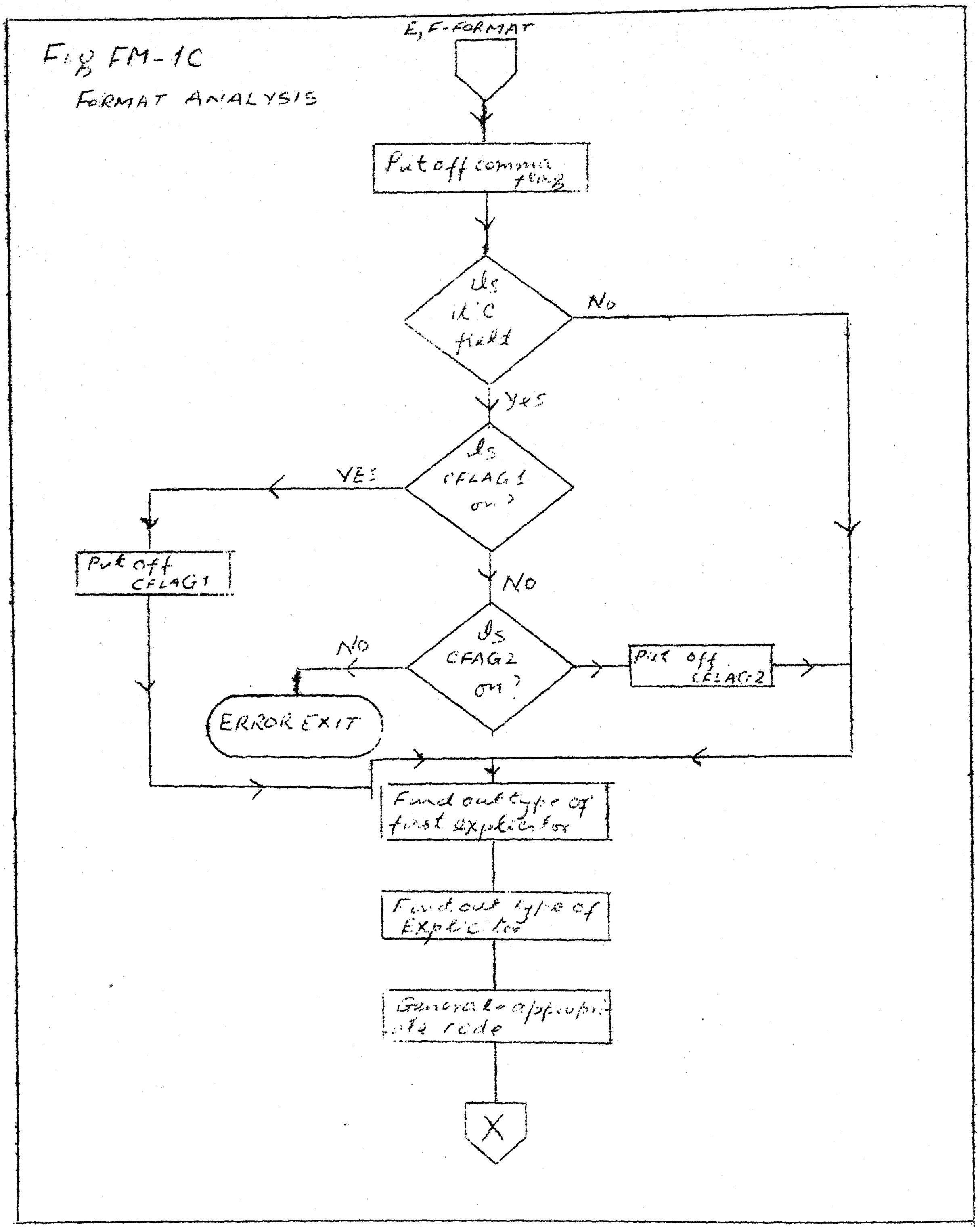
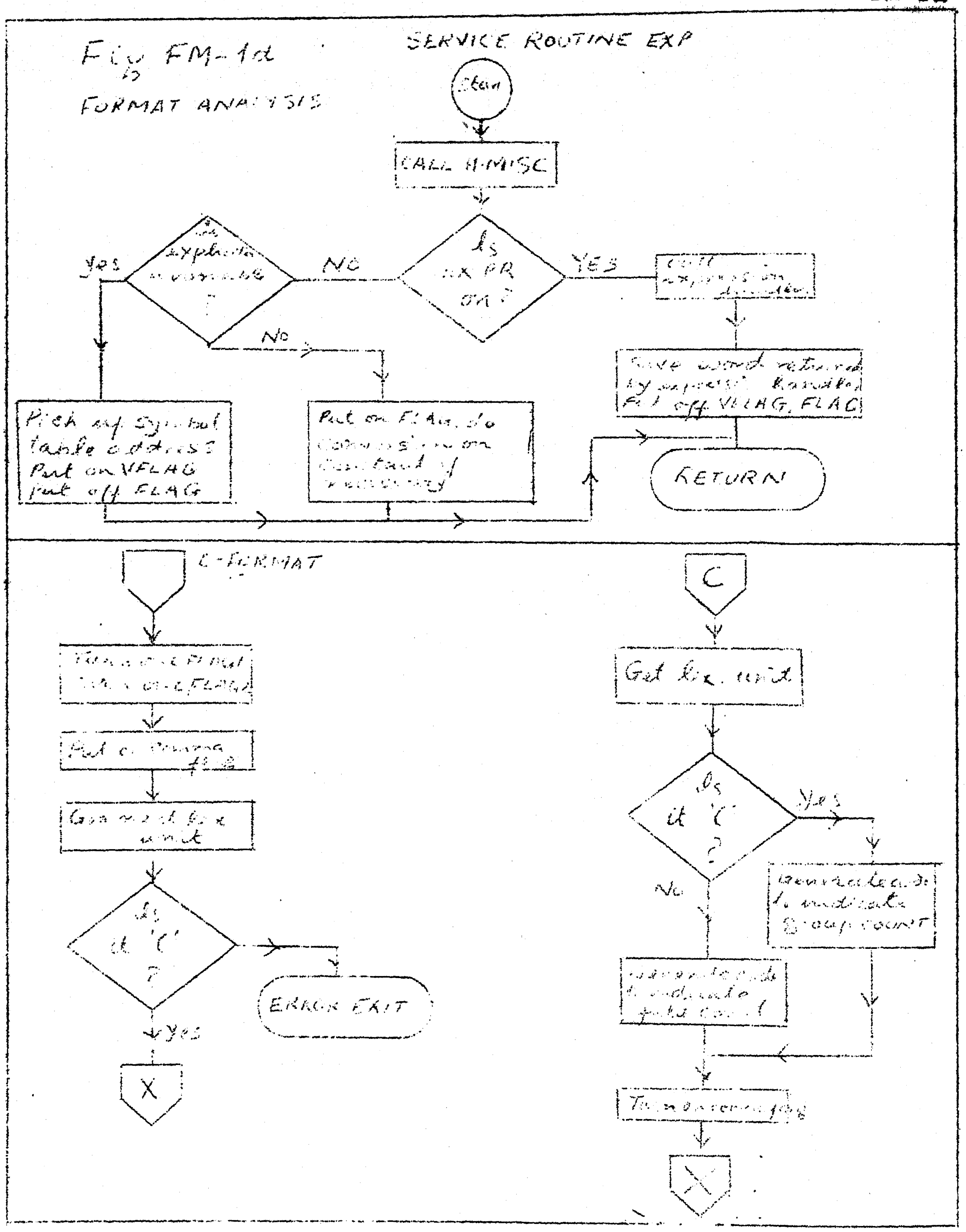


Fig FM-1d

FORMAT ANALYSIS


SERVICE ROUTINE EXP
Start
CALL H-MISC
Is EXPR on ?
YES
call expression [illegible]
Save word returned by expression handler Put off VFLAG, FLAG
RETURN
NO
Is explicit [illegible] ?
Yes
Pick up symbol table address Put on VFLAG put off FLAG
No
Put on FLAG, do conversion on constant [illegible]
C-FORMAT
[illegible] FLAG
Put on [illegible] flag
Get next [illegible] unit
Is it '(' ?
Yes
ERROR EXIT
X
C
Get [illegible] unit
Is it '(' ?
Yes
Generate code to indicate Group count
No
Generate code to indicate field count
Turn on [illegible] flag
X

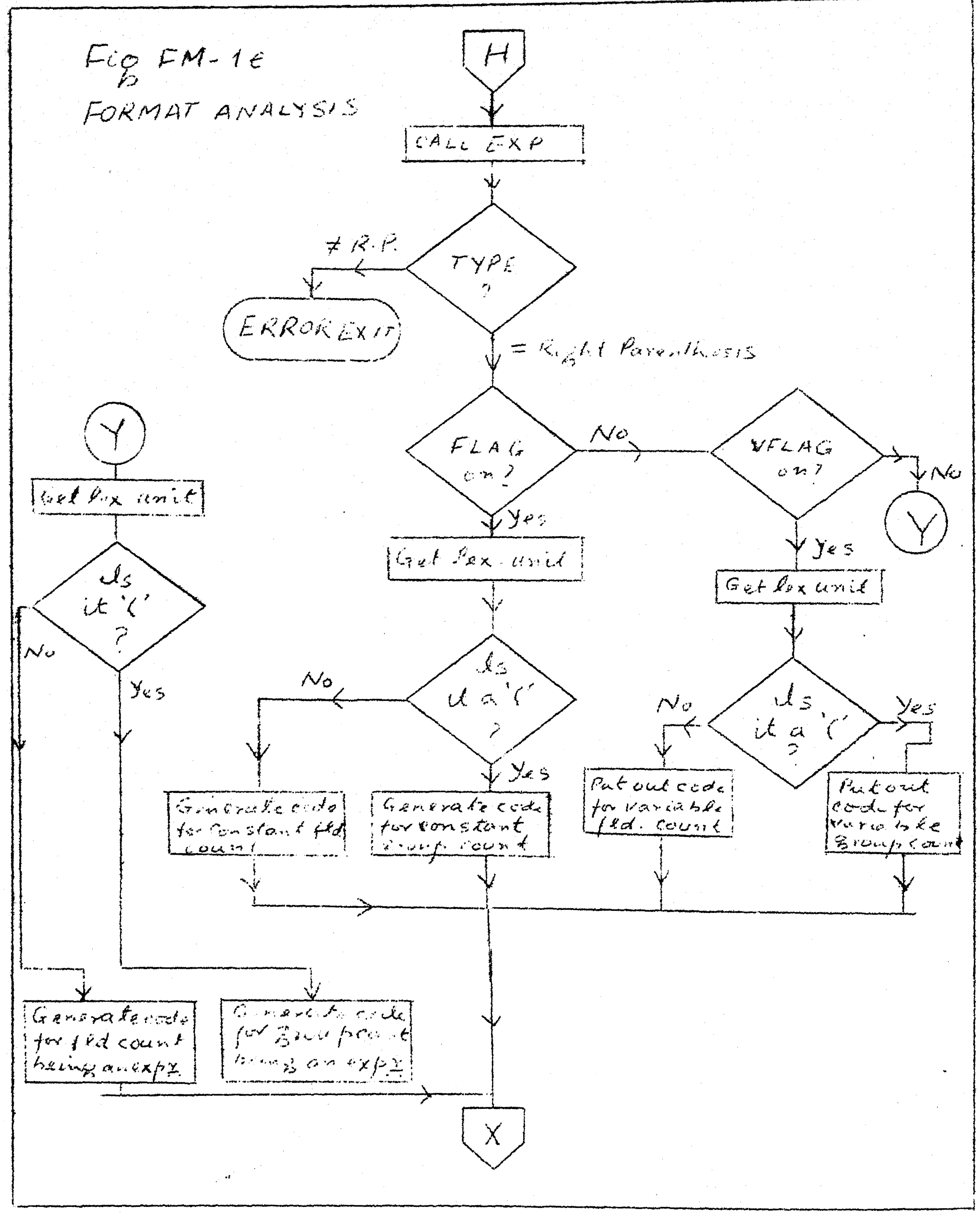
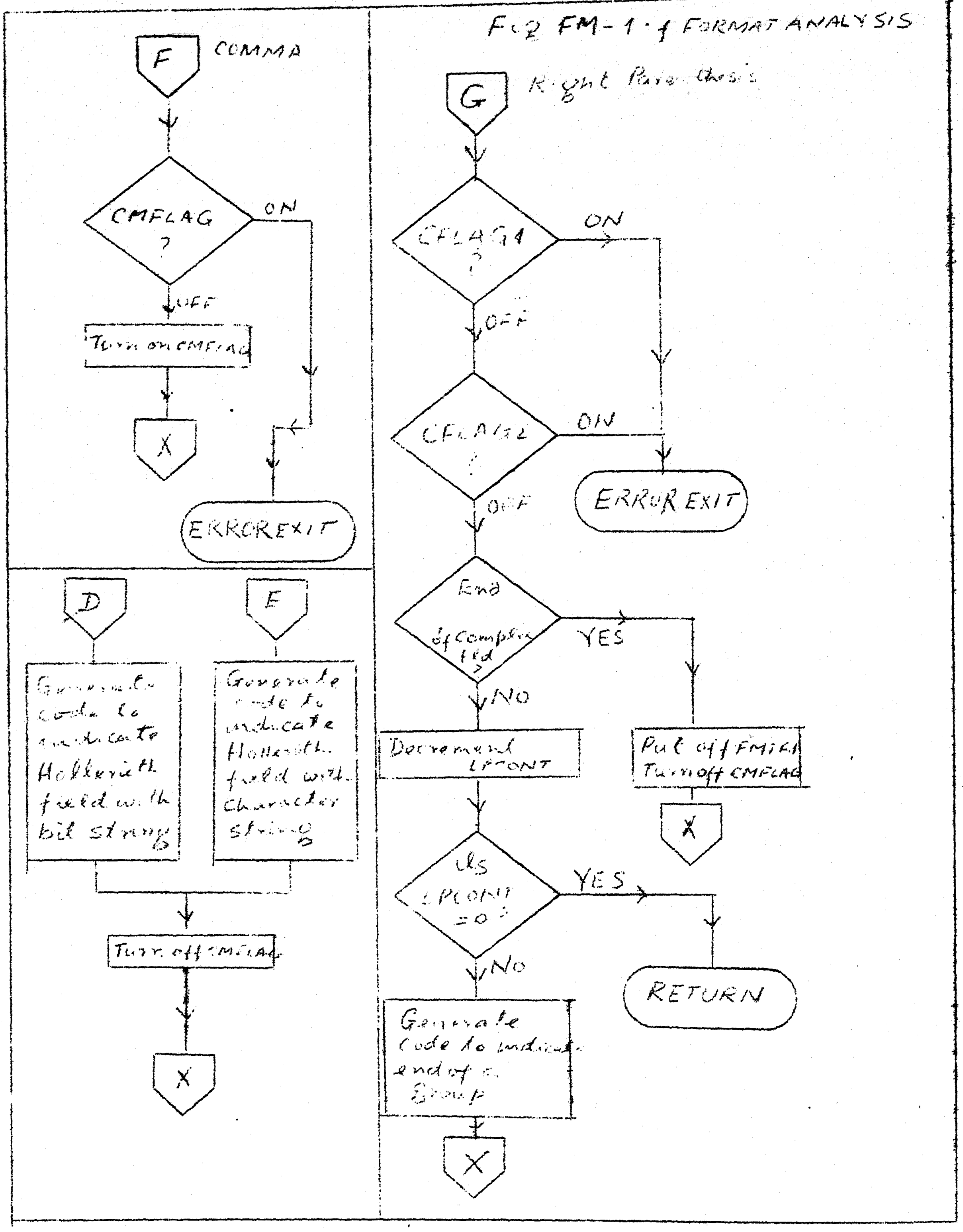


Fig FM-1e
FORMAT ANALYSIS
H
CALL EXP
TYPE ?
≠ R.P.
ERROR EXIT
= Right Parenthesis
FLAG on?
No
Yes
VFLAG on?
No
Yes
Y
Get lex unit
Get lex unit
Get lex unit
Is it a '(' ?
No
Yes
Generate code for constant fld count
Generate code for constant group count
Is it a '(' ?
No
Yes
Put out code for variable fld. count
Put out code for variable group count
Y
Get lex unit
Is it '(' ?
No
Yes
Generate code for fld count being an expr
Generate code for group count being an expr
X

Fig FM-1·f FORMAT ANALYSIS


F
COMMA
CMFLAG ?
ON
OFF
Turn on CMFLAG
X
ERROREXIT
D
E
Generate code to indicate Hollerith field with bit string
Generate code to indicate Hollerith field with character string
Turn off CMFLAG
X
G
Right Parenthesis
CFLAG1 ?
ON
OFF
CFLAG2 ?
ON
OFF
ERROR EXIT
End of Complete fld ?
YES
NO
Decrement LPCONT
Put off FMIFL Turn off CMFLAG
X
Is LPCONT = 0 ?
YES
NO
RETURN
Generate code to indicate end of a group
X

CHAPTER VI

RUN TIME I/O ROUTINES

1. Introduction:

Run time routines required to support PL/I I/O commands carry out the following tasks:

a) Supervising file operations.

b) Simulation of stream oriented I/O.

c) Providing free format I/O (List directed I/O).

d) Providing editing facilities other than those in Fortran.

e) Providing the facility of using data names in I/O streams.

f) Initiating error recovery procedures f I/O operations.

In addition to routines which were written to perform the above tasks, a number of modifications were incorporated in existing Fortran I/O routines in order to a) to take care of different addressing schemes used in PL7044 and b) to conform to the stream orientation of PL 7044 I/O. These routines occupy about 4 K locations in memory excluding the space required by certain run time table which form a part of the object code.

In the following pages, the routines are discussed under the respective tasks performed by them.

2. Supervising File Operations:

In PL 7044 there are two main types of file organisation- the stream oriented and the record oriented file organisations.

The stream oriented files can either be input or output files while the record oriented files can be either input or output or update files. A stream oriented file cannot be used for record oriented I/O and similarly a record oriented file may not be used in stream oriented I/O. Further, an input file may not be used for an output operation nor can an output file be used for input operations.

Finally, an update file may not be used in a stream oriented I/O command. These restrictions require that every I/O command be validated before it is executed. One would think that such validation may be carried out at compilation tune and thus save time during execution. But this is not always possible since file names, just as any other variable names, may be passed on as arguments of a procedure reference, in which case validation, perforce, will have to postponed till execution time. In order to ensure uniformity of treatment of normal files and formal files, all validation operations are undertaken at run time. This simplifies the logic at compilation time although one would have to pay for it in increased execution time.

The Need for Saving the Status of a File:

I/O Operations are carried out on different files at different times by a collection of run time routines. A file control block is maintained by IOBS for every file. This block contains information regarding record length, buffer address etc. However, it does not contain certain information necessary for carrying out stream oriented I/O and identification information. The precise nature of information not available in the File Control Block are:

i) Total number of records written into/read from a file.

ii) Type of file (Input, output, print, stream, record etc.).

iii) Word and character pointer in a particular data record.

iv) Indications regarding end-of-record condition.

Such information as above are usually imbedded in the routines that simulate the stream I/O. Therefore, when we switch I/O operations from one file to another it is necessary to save the above information of the first file and restore the information corresponding to the second file.

In order effect such saving and restoring, the above information resides in what is known as the file status block. At the initiation of an I/O command, the file that is to be activated is checked with the file that was involved in the last I/O operation. If the two check, no saving or restoring is done. But if the two files donot check, then the status of the file last involved in an I/O operation is saved in its file status block, and the status of the file on which a new I/O operation is to be performed, is restored in the corresponding locations of the I/O routine. Thus three distinct operations must be performed before an I/O operation is allowed to take place, namely - validation, saving and updating.

The routine which performs all these functions is known as the I/O supervisor and is called before each I/O command is executed. The calling sequence for this routine is,

```
TSX  .IOSUP,4
PFX  fcb,,fsb
```

where,

fcb	=	address of the file control block of the file in question.
fsb	=	address of the file status block.
PFX	=	PZE for stream input
	=	PON for stream output
	=	MZE for record input
	=	MON for record output.

It may be noted that while the file control block is created by IBLDR from information available in the file declarations, the file status block is created for the files by the compile time routines of Pass 2. The format of the file status block is given in Appendix K

Description of the I/O Supervisor:

Figure RN-1a gives the flow chart of the I/O supervisor routine. First the information regarding the type of file is accessed from the file status block. This is matched with the type of operation requested in the calling sequence. If the match fails, error procedures are initiated. Next, the nature of the last file activity is examined. If it indicates a stream oriented I/O operation, and if the current request is

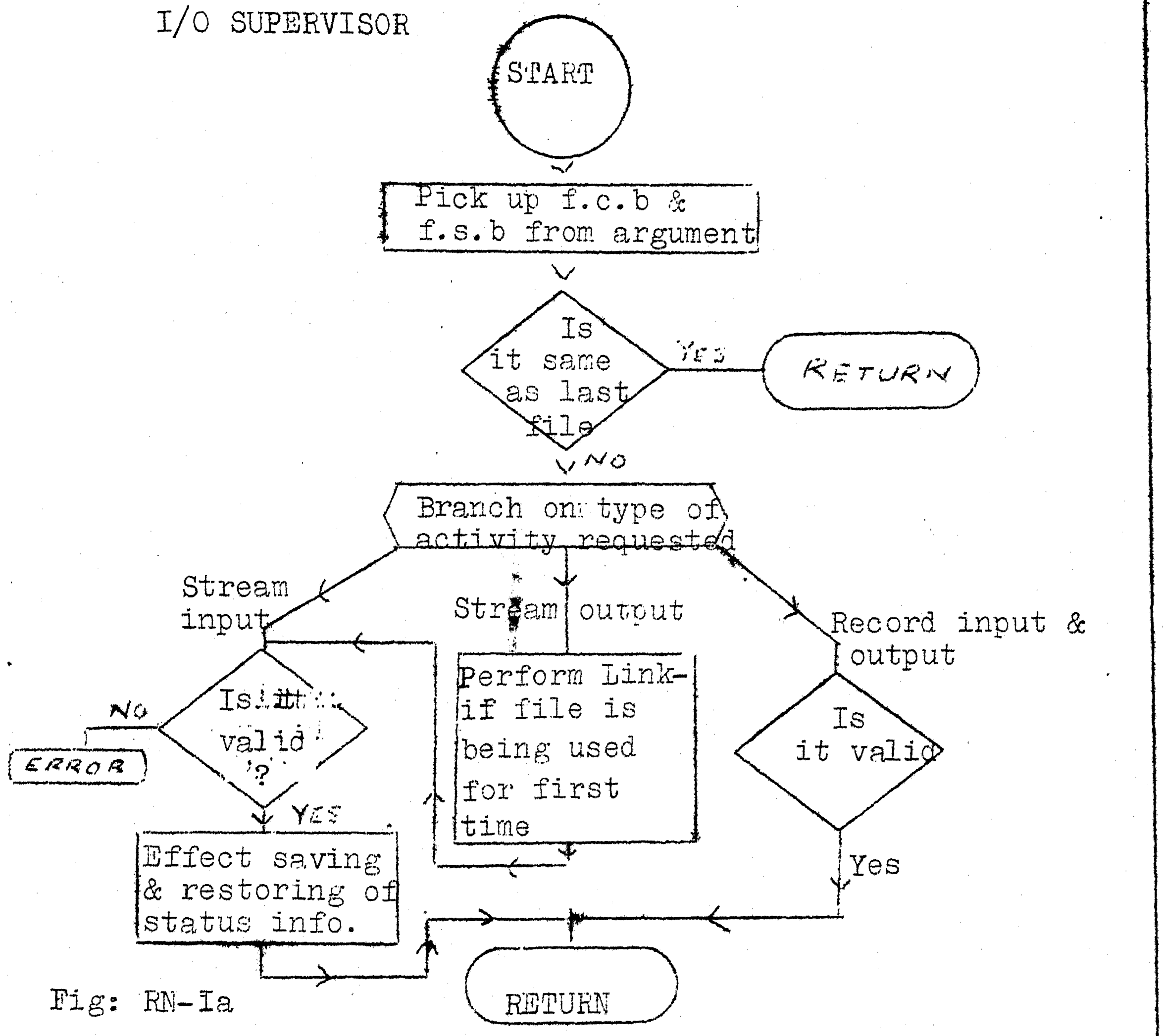
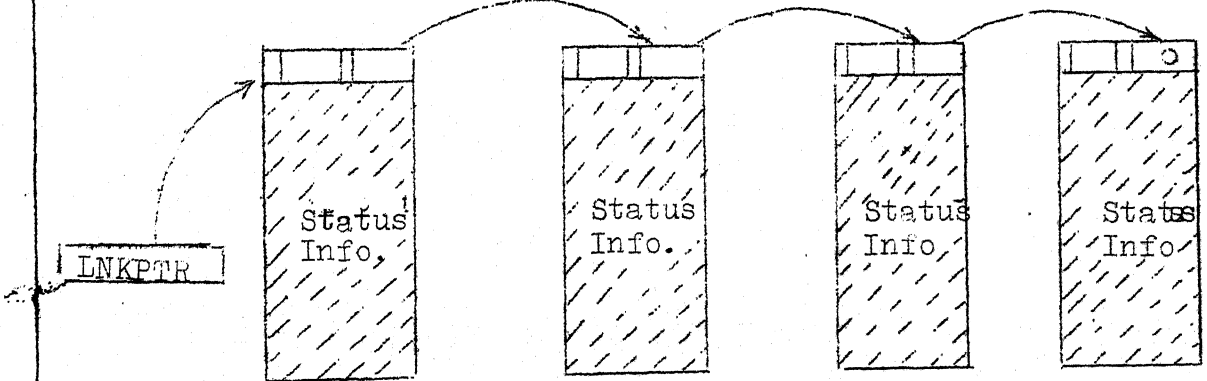

I/O SUPERVISOR
START
Pick up f.c.b & f.s.b from argument
Is it same as last file
YES
RETURN
NO
Branch on type of activity requested
Stream input
Stream output
Record input & output
Is it valid ?
NO
ERROR
YES
Effect saving & restoring of status info.
Perform Link- if file is being used for first time
Is it valid
Yes
RETURN

Fig: RN-Ia



LNKPTR:Link Pointer

Fig. Ib : Linking of output Files

also for a stream I/O operation, then the information regarding the status of the file is saved in the file status block of the corresponding file. At this point it is necessary to point out that all output files (stream or record oriented) are chained, the link being formed when the file is used for an I/O activity for the first tune. This link is created in this routine. The address part of the first word of the file status block is used for linking. The purpose of the linking is to enable the error recovery routine to flush the buffers of the output files one after another, thereby preventing any loss of information when an error condition is raised. The sign bit of the last word of the f.s.b is used to indicate whether a file was involved in any I/O activity.

After the saving operation,comes the restoration, if necessary. The f.s.b of the file on which I/O has been requested is consulted to check if it contains useful information. If it does not, the file is opened by appropriate calls to IOBS and the locations in RWD deck (described in the next section) are set to certain standard options and an exit from the routine is taken. If however, the f.s.b contains useful information, then this information is made use of for updating the locations in RWD deck. The details of the updating action are as follows: for an input file, a pointer is set to read a new record if the file is being used for the first time, or certain other pointers are set to read from where the last input activity on the file ceased to read. The set of pointers involved are the character pointer, the word pointer, the read-a-record indicator, the system input file indicator, the address of the word in buffer and the counter indicating the total number of records read from the file. For an output file, the updating action is as follows: if the file has not been used previously, first the linkage is established as discussed earlier. This linkage is shown pictorially in Fig. RN-1b. Then the file is opened, and space in the buffer is requested from IOBS. The address of the space located by IOBS is used to reset the word pointer in deck RWD. Note that the word pointers for input and output are different locations although they belong to the same deck RWD. If the file has already taken part in an output activity, then information in the f.s.b. is used to set the word pointer to write from a point in the output record where the last output activity on that file ceased to write. The other indicators which are restored are the pending word, the counter indicating the total number of records written on the file and the number of characters available in the pending word for output.

It may be noted in passing, that no distinction is made between system input and system output files on one hand and other stream oriented files on the other since the validity, saving and restoring operations are carried out on all files.

In case of record oriented files - the I/O supervisor returns after the validity checks have succeeded. This is because record files do not require saving and restoring of status information as in the case of stream files. This is, in turn, because record files are analogous to Fortran binary files and each I/O command implies that a new record is to be read or written.

3 Routines to Simulate Stream Oriented I/O:

The routines which are responsible for simulating stream oriented I/O of PL 7044 are all located in a deck called the RWD deck. Before discussing the functions and 'modus operandi' of each routine in this deck it is best to set out the names and functions of the more important variables and entry points in this deck.

Notations:

BU1.	Entry point to advance output buffer when ACC is full.
BU4.	Entry point to advance output buffer when MQ is full.
CCA.	Entry point to save pending word if it is in ACC.
CCQ.	Entry point to save pending word if it is in MQ.
CHRPTR	Character pointer.
DATUM	Holds address of I/O data item.
FCBLOC	Address of file control block.
FMTLOC	Format location
GETCH1	Address of word in input buffer from which the next character is to be obtained.

GLTCH2	Conditional branch instruction for reading the next record.
GETCH.,GETCH	Routine which gets a character from the buffer area.
HCT.	Final entry point from format conversion routines.
HNLIO.	Intermediate entry point for edit directed I/O.
IOHCM.	Initial entry point from format conversion routines.
IOHCT.	Final entry point from conversion routines.
IOHEF.	End-of-Format entry point.
IOHLP.	When left parenthesis is met in format.
IOHRP.	Entry point for right parenthesis in a format statement.
IOSW	Switch to indicate Input or output.
LINENO	Gives the number of records written in the current file.
NEWTOP	Routine to push down format return address in format stack.
POPTOP	Routine to pop off one item from the top of the stack.
RESET	Entry point for initializing output buffer.
IOHIO.	Entry point for reading/writing a new record.
SC.	Entry point to update XR1 and XR2 prior to format conversion on output files.
STHIO.	Initial entry point for output edit statement.

TSHIO.	Initial entry point for input edit statement.
SUBSTP	Routine to change value of the top of the format stack.

The routines which are located in the RWD deck can be grouped together according to the following subtasks.

1) Routines to perform linkage between format list and the data list.

2) Routines to perform reading or writing of data.

3) Service Routines.

1) Routines to perform linkage between format list and data list. Under this category the routines included are:

(a) TSHIO.	(b) STHIO.
(c) IOHCM.	(d) HNLIO.
(e) IOHLP.	(f) IOHRP.

The routine TSHIO. is called by the object code and has the following calling sequence,

```
TSX  TSHIO.,4
PZE  <format list address>
```

The GET EDIT command is initiated by a call to this routine which picks up the address of the format list from the argument and transfers control to the first instruction of the format list after initializing certain pointers and saving the return address of the caller. The format list is a sequence of calls to various conversion routines. The conversion routine, if it is not a control format conversion routine, will call back at any entry point IOHCM. in the RWD deck. This routine updates certain counters like the field width etc. with the help of the argument in the calling sequence to the conversion routine. IOHCM. then transfers control back to the object code. The object code then loads accumulator with the address of a variable that is to be inputted or outputted. Then a call to HNLIO. is given. This routine saves all index registers in preassigned locations and calls back the conversion routine at the appropriate point. (Every data conversion format routine has two entry points apart from the main entry point: one for input and the other for output). The conversion routine does the required conversion and hands back control to IOHCT. This routine, after ascertaining whether the same conversion routine is to be repeated or not, returns control to the format list. If the format item is to be repeated, then control is not returned to the format list but to the object Program. Fig. RN-2 depicts the path of control schematically.

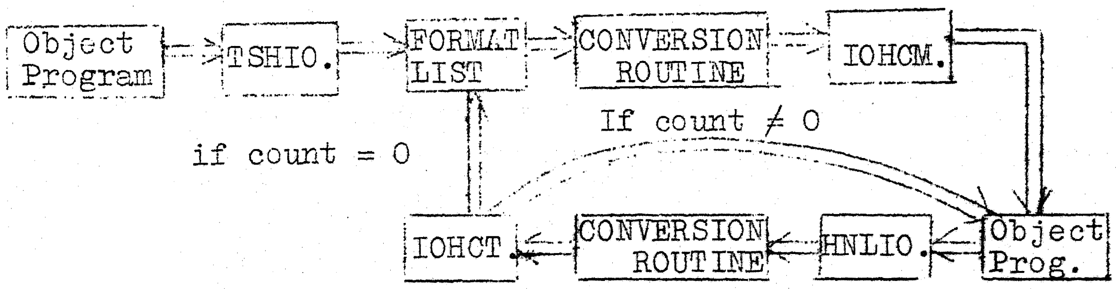


FIG. RN-2.

Two exceptions are made to the above path of control. The first exception is made when the format list contains a reference to the opening or closing of a group and the second exception is made when a call to a control format routine is executed.

(a) When a format list has a reference to a group opening (IOHLP.), the point from where the call is made and the group count is saved in a push down stack. This is done by calling the routine NEWTOP. In the case when the format list has a reference to a group closing (IOHRP.) the top most entry is looked up and the return address contained in that entry is used for transferring control. Before this, however, the group count is reduced by 1 and stored back. If the group count has been reduced to 1 then the corresponding entry is popped off and the next entry is given the same treatment. The routine to reduce count is SUBSTP while the routine to pop off the top of the stack is POPTOP. The stack looks as follows:

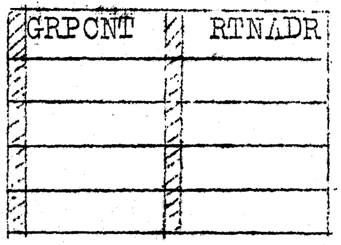


FIG. RN-3

where,

GRPCNT = group count and

RTNADR = return address

The maximum depth to which the stack can be filled is limited to 10. The need for a stack arises because of the fact that in PL 7044 groups may be nested, and the expectation is that each member of the nest will be fully satisfied before traversing the higher member. In Fortran, however, the nested groups will be traversed once and from there onwards only the inner group will be repeatedly traversed till all data items are exhausted. To illustrate this point consider the following format statement:

```
Fortran - FORMAT (5(2X,3I2, 7(10X, 8(I5,5(10X,Ø12)))))
PL 7044 - FORMAT (5(X(2),3I(2),7(X(10),8(I(5),5(X(10),Ø(12)))))).
```

The object code in the two case will be identical but the interpretation given by the two languages will be different. Whereas in Fortran the information regarding the first 3 group counts will be 'lost', in PL 7044 they are saved in the push down stack mentioned earlier. When the innermost group has been traversed 5 times the next group of 8 is taken up and this continuous till all the groups have been traversed. The outermost pair of parenthesis is assigned a group count of 32767 so that this group count can never be exhausted in a practical case.

(b) When the format list has a call to a control format editing routine, then there is no need to refer back to the object code. The need will arise only if a data item is to be inputted or outputted. The routines which do not refer back to the object code are page skipping routine, line skipping routine, hollerith conversion routine, column skipping routine and X-format routine.

In all the above cases, the path of control will be as follows:

$$\begin{tikzcd}\boxed{\text{Object Code}} \arrow[r] & \boxed{\text{TSHIO.}} \arrow[r] & \boxed{\text{Format List}} \arrow[r] & \boxed{\text{Conversion Routine}} \arrow[l, bend right]\end{tikzcd}$$

FIG. RN-4

Just as TSHIO. was used for inputting variables the routine STHIO. is used for outputting data items. The calling sequence is identical to that of TSHIO.

2) <u>Routines to perform reading or writing of data</u>:

Let us now discuss the routines which are used for inputting or outputting data items by the stream directed I/O commands. The routines which fall under this category are:

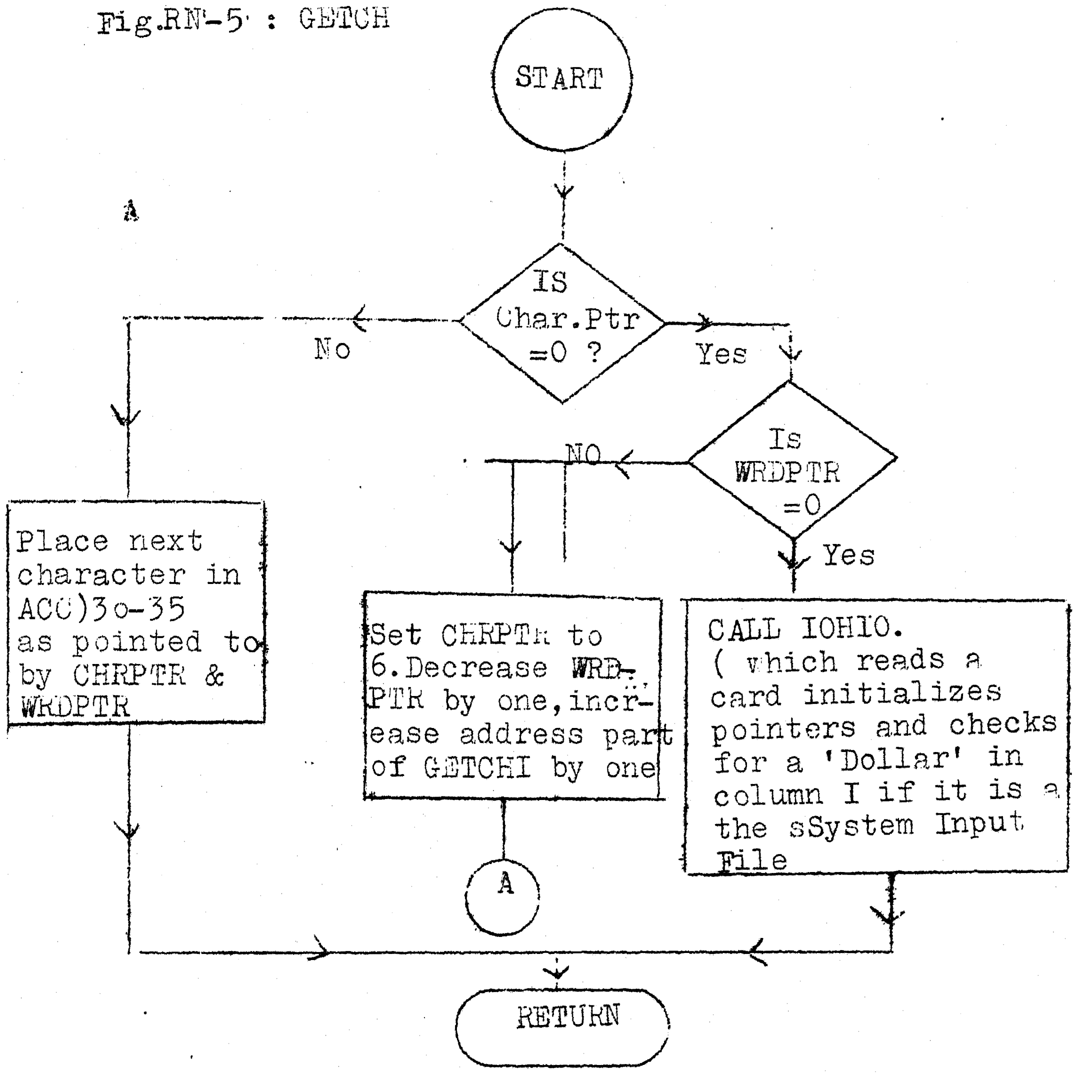
(a) GETCH

(b) IOHIO.

(c) BU2.,BU4.

(a) Fig. RN-5 gives the flow chart of the routine GETCH. This routine when called for the first time for an input stream file, initiates a call to IOHIO. to read a data record from the file. IOHIO. puts thru a call to IOBS to locate a record in the buffer. IOBS then passes on the address of the record so located. IOHIO. then stores this address in GETCH1, turns GETCH2 positive, initializes word pointer to the number of words in the data record, character pointer to six and returns control to GETCH if it is not the system input file. If it is the system input file, then IOHIO. checks for a '$' in the first character position of the first word of the data record. If a '$' is encountered, the

Fig.RN-5 : GETCH


START
IS Char.Ptr =0 ?
No
Yes
Is WRDPTR =0
NO
Yes
Place next character in ACC)30-35 as pointed to by CHRPTR & WRDPTR
Set CHRPTR to 6.Decrease WRD-PTR by one,incr-ease address part of GETCHI by one
CALL IOHIO. (which reads a card initializes pointers and checks for a 'Dollar' in column I if it is a the sSystem Input File
A
RETURN

end-of-data error procedure is initiated. When GETCH routine gets back control it loads XR1 with the character pointer and by means of a table look up, places the appropriate character in ACC)30-35 and returns. If the file is not being used for the first time or when the end-of-record indicator is not on, then the call to IOHIO. is not made but the remaining part of the logic remains as in the above case. In this routine there are a number of locations which need to be saved in the File Status Block described earlier. These locations are:

RECNO	number of records read from the file
WRDPTR	word pointer
CHRPTR	character pointer
SIGN	system input file indicator
GETCH2	end-of record indicator
GETCH1	address of the word in buffer currently being used in character access.

(b) IOHIO: The function of this routine in relation to an input operation was discussed in the last section. This routine is also called by the output routines BU2. and BU4. IOHIO. differentiates between the two calls (from GETCH. and BU2./BU4) by the setting of the switch IOSW. If it is +ve, then the call is for inputting a record, if it is -ve it is for outputting a record. When the call is for outputting a record, IOHIO acts in the following manner. It first checks if at least 3 words have been written in the line to be outputted. This is because a record may not have less than 3 words in it. (Hardware restriction). If the number is less than 3, it pads up the remaining positions with blanks and then informs IOBS that a line has been written in the buffer. It also passes on the number of words in the record as an argument. It then initiates a call to RESET which resets the pointers and readies the next record for writing. On return from RESET, IOHIO. returns control to the calling routine.

(c) BU2. and BU4: These two routines are used by the conversion routines to place the contents of the accumulator or of the MQ register in the output buffer. The buffer address is available in XR2 in complemented form. When the value of XR2 exceeds the address of the last word in the data record, a call to IOHIO. is initiated which locates the next record and returns.

<u>Service Routine</u>:

The routines that fall under this category are:

a) RESET

b) SC.

c) CCA/CCQ.

d) XC.

a) <u>RESET:</u>

This was already mentioned in connection with IOHIO. in brief. Its calling sequence is TSL RESET. It performs a call to IOBS for locating space for the next output record. It then resets pointers before returning.

b) SC.:

This routine is entered by every conversion routine to update XR2 with the address of the word in the current data record, to load XR1 with the number of characters available in the pending word and to load the accumulator with the pending word itself.

c) CCA. and CCQ:

These two routines are entered when the conversion routines have finished their job and the pending word is left in the accumulator or the MQ register. The CCA. is called when the accumulator contents are to be saved and the CCQ. is called when the MQ is to be saved. Besides this, the routines also save the contents of XR1 and XR2 in preassigned locations from where SC. can load them back when SC. is called by the conversion routines.

d) XC.:

This routine is called by the conversion routines whenever blanks have to be created at the beginning of a field. For example, if a number occupies 4 characters and if the format specifies a field width of 8, then 4 leading blanks have to be created. This is done by calling XC. with the XR4 containing the number of blanks to be created.

In SC. there are four words that need to be saved when I/O commands necessiate a switching of files. These locations are:

(i)	CHRPTR	No. of characters in pending word.
(ii)	WRDPTR	Address of data record word in 2's complements.
(iii)	BU22.	Pending word.
(iv)	LINENO	Number of records written in the file.

These words are saved in f.s.b by the I/O supervisor when it detects a change in the file being used.

4. Routines to Provide Free Format I/O (List Directed I/O):

The routines which fall under this category are:

a) GETITM

b) G.LIST

c) P.LIST

a) GETITM: This is the basic routine for free format conversion. A call to this routine takes the form TSL GETITM. This routine collects a data item from the input file currently in active state. This data item will be henceforth referred to as the source. Depending on certain indicators set by the superior routines, GETITM performs the necessary conversion of the source to match the attributes of the variable to which the source is to be assigned. This latter entity will be called the target in subsequent discussions. Not in all cases will the conversion of source to target succeed. For example, if the target is a floating point variable and the target is a character string data item having non numeric characters, conversion will fail. In such cases, processing is terminated after a suitable error message is written out on the system output file. The table of conversion is given in Appendix M3.

<u>Description of the Routine</u>:

<u>Notations</u>:

MFLAG ... Mode flag

= 0 for integer target

= 1 for floating point target

= 2 for character string target

= 3 for bit string target

= 4 for complex target

(The mode flag is set by the superior routine depending on the nature of the target)

SINFLG = +ve if the source is a positive arithmetic constant

= -ve if the source is a negative arithmetic constant

(SINFLG is used to set the sign of the source after conversion has been performed.)

SFLAG = -ve for a 'null' item

ITEM1 = Contains the converted source if real arithmetic.

ITEM2 = 0 for real values

≠ 0 for complex items

(for complex data items ITEM1 will contain the imaginary part and ITEM2 will contain the real part).

COMFLG	Flag to indicate that the source is a complex data item.
BITEM	Contains Binary items.
COUNT	Number of characters in a data item. This required for targets which have the character attribute.
FCOUNT	Count of characters after decimal point. (required during conversion).
IOBF	Buffer to collect the characters if target is a character variable.
NQFLAG	-ve when source is a character string and target is numeric.
QFLAG	-ve if source is a character data item.
QUOTE1	-ve if target is arithmetic.
DOT	-ve if '.' is to be treated as a character.
DOT1	-ve if '.' has occurred once in source.
REAL	-ve if target is real.

(REAL is set by the superior routine for all real targets).

LSTCHR	-ve if it contains a character which must be examined before fetching another character from the input file.

Figures RN-6a to 6e give the flow chart of the GETITM routine in 5 parts. Part 'a' is the main branch of the program. The decisions, in this part, are taken with the help of a 64-entry table, each entry corresponding to one character in the IBM 7044 character set. This table is also used by the other parts of the routine. Part 'b' is concerned with the collection of arithmetic source items except those with exponents. Part 'c' is concerned with the collection of character source items. Part 'd' collects source arithmetic items when specified with an exponent. Part 'e' performs the necessary source to target conversions.

(b) The routine G.LIST directs the GETITM **routine** to collect and convert the data item from the input file and place it in preassigned location(s). This routine then transfers the data item so collected to the location(s) whose address(es) is (are) supplied by the object code. The calling sequence of this routine is,TSX G.LIST, 4

```
          TSX G.LIST, 4
          PFX A,,B
```

where,

PFX	=	PZE for integer target,
	=	PON for floating point target,
	=	PTW for bit string target,
	=	MZE for character string target,
	=	MON for complex target,
A	=	Address where converted source is to be stored if target is an integer, floating point or complex item.
	=	Address of bit string if target is specified as bit string. The length specification is to be stored in the next location.

= Address of the header word if the target is a character scalar,

= Address of word in run time stack if target is character array element,

B = Address of picture string.

(In the present implementation, no provision has been made for handling picture strings).

Figure RN-7 gives the flow chart for this routine. As can be clearly seen, five different sets of action have to be taken corresponding to five distinct target items. Each set of action prepares the ground for calling the routine GETITM by setting the appropriate indicators and stores the data item collected in the respective addresses supplied by the object code. In case of character and bit string targets, besides storing the data item, the length specification is also updated if the string is declared to be a varying string. If the length is fixed, then the data item collected may undergo truncation or padding with either blanks (for character string) or zeros (for bit string), depending on whether the target length is less or greater than the source length specification.

(c) P.LIST is a routine similar to G.LIST having the same calling sequence, but is used for outputting data items in free format. This routine does not have its own conversion routines but uses conversion routines used by EDIT I/O commands. Format lists used by P.LIST are standard format lists for each type of data item to be outputted. In case of character and bit strings the format list is modified (before calling the appropriate conversion routine) to accommodate exactly the number of characters to be outputted. In case of arithmetic data items, if there is no space available in the current record to accommodate all the characters, that record is skipped and the characters are written out on a new record. The formats used for each type of data item is as follows:

Integer	(X(2), I(11))
Floating Point	(X(2), E(14,8))
Complex	2(X(2), E(14,8))
Bit String	(X(2), B(*))
Character String	(X(2), A(*))

indicates variable length.

5. Routines for Providing Data-Directed I/O:

The routines which fall under this category are:

a) G.NAME

b) R.DATA

c) P.DATA

a) G.NAME:

This routine reads a name from the input file and saves it in a preassigned area. The length of the name read cannot be more than 30 characters. This routine is always called by R.DATA and has the calling sequence TSL G.NAME.

b) <u>R.DATA:</u>

This routine is responsible for inputting data items whose names are specified along with their values. The format of the input stream is as follows:

NAME1 = ..., NAME2 = ..., NAME3 = ... $

The calling sequence for this routine is,

```
TSX  R.DATA,4
PZE  n
PFX  M1,,N1
PFX  M2,,N2
 .
 .
 .
PFX  Mn,,Nn
```

where,

PFX	=	PZE for scalar items
	=	MZE for dimensional item (arrays)
M_i	=	Address of i-th entry in name table,
N_i	=	Address of word in the i-th entry from where dimension information starts,
	=	0 for scalar items,
n	$\neq$	0 for arrays and cross section of arrays,
n	=	number of variables appearing in the GET DATA statement.

The flow chart for this routine is shown in Fig. RN-8 . The contents of the name table entry have been shown in the Chapter on Implementation details.

Routine Description:

It picks up the number of data variables from the argument list and calls G.NAME which collects a name from the input file. This name is matched against the names stored in the run time table entries M_1 thru M_n. If there is no match, an error condition is raised and the job is terminated.

If there is a match at M_k, then that entry is further examined for its type, address and number of dimensions (if the calling sequence indicates that M_k has dimensional information). If it is not a dimensioned variable, the routine looks for an '=' in the input file. If it is a dimensional variable, then the subscript value is looked up from address N_k. If the subscript value is $(77777)_8$ then it signifies that subscript must be read from the input file. The subscript (s) must be enclosed in parentheses. Let us illustrate this with an example:

```
GET DATA (A(3,4,*), B,C);
```

A is a 3 dimensional array, In the data list two of the subscripts are constants while the 3rd one is varying. The corresponding input stream will be,

```
A(1)  =  10, A(2)  =  3 , ... A(10) = 25.0 ... $
```

Note that although A is a 3-dimensional variable, in the input stream only one dimension is specified. This is because it would be meaningless to specify the other two which are known to be constants. The run time table for A will be as follows:

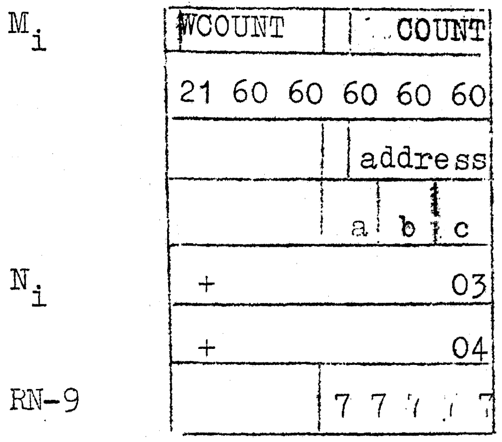


Fig.RN-9

WCOUNT	= 1
COUNT	= 1
a	= 3 (no. of dimensions)
b	= 1 (minor type)
c	= 0 (major type)
address	= header address of A

In the above table $(77777)_8$ indicates that the 3rd subscript is to be read from the input file.

After collection of subscripts, if any, this routines calls the address calculation routine ADDCAL for calculating the address of the array element. The address returned by this routine is then plugged in as an argument in the call to G.LIST. Alternatively if the variable in question is not an array element, the address available in the name table in placed in the argument list of the call to G.LIST. G.LIST then collects the data item from the input file and transmits it after due conversions, to the proper address(es).

Now the above discussion pertains to one cycle in this routine. The cycle is repeated until an end-of-data-set is encountered on the input file. The end-of-data-set symbol is a '$' in PL 7044. Note that this '$' should not occur in Col. 1 of a data card, for this will give rise to an end-of-data condition. Thus a data set may look like the following.

A(1) = 1, B = 6, A(3) = 5$ for the example given earlier. Note that the data set need not appear in the same sequence as in the data list, nor is it necessary for all of the variables in the data list to appear in the data set. However, a variable name appearing in the data set must have appeared in the data list, otherwise an error condition will be raised.

c) P.DATA:

This is a routine similar to R.DATA except that it is responsible for outputting data directed variables. The calling sequence is ~~x~~identical to that of R.DATA. This routine also makes use of the run time table entries supplied to it by the object code and outputs the names and their values in the order in which they appear in the argument list. Unlike R.DATA, however, there is no search involved here since all the variables have to be outputted. When an array cross section is involved, then P.DATA has a built in loop which **traverses** the entire cross-section of the array and outputs each element with the name and subscripts appearing on the left hand side of an '=' sign. The value appears on the right hand side.

Example: PUT DATA (A(4,*,5),B,C); will result in

```
        A(4,1,5) = ...  A(4,2,5) = ...., A(4,3,5) = ...
        ...  ...  ...  ...  ...  ... A(4,N,5) = ...
        B = ..., C = ... $
```

of the first word of the new record, is updated with the help of the word pointer which was initially saved. The character pointer is also updated with the help of the old character pointer. By doing this, the relative position in the new record w.r.t the old record is maintained. If the current file is an output file, a similar set of n calls to IOHIO. ensures that n blank records are written out. As in the input case, the relative positions in the old and new records are made to agree.

Thus, for example the action of SKIP (4) on an input file is shown in the figure below:

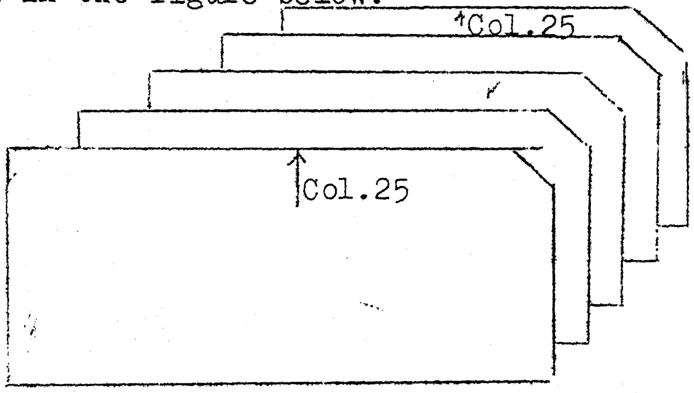


FIG. RN-10

The two arrows show the position before and after the SKIP command.

(b) <u>LCV:</u>

LCV is the run time routine which executes the command LINE(n). It is similar to a skip command, except that instead of skipping n records, it skips to the n-th record of the file if possible. If n records have already been processed, then no action is taken by this routine. No diagnostics will be given either.

(c) PCV:

This routine executes the PAGE command. It is similar to the format 1H1 in Fortran but can be given along with the PUT statement itself or in the format list. Also if the file being processed has an attribute other than the PRINT attribute an error condition will be raised. The calling sequence for this routine is,

```
TSX  IOHPC.,4
PZE  0
```

The argument is redundant and is present only for the sake of uniformity with other format routines which have one argument.

(d) CCV:

This routine handles the command COLUMN(n). The calling sequence of this routine is,

```
TSX  IOHCC.,4
PZE  n
```

where n is the column to which the position pointer in the record must be moved. If the current position is greater than n then the routine will skip to the first data position of the next record. This is valid for both input and output records.

Example: Consider the system input file:

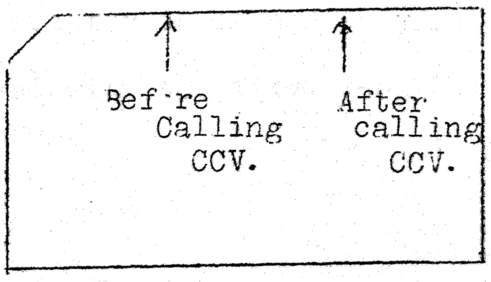


Fig. RN-11

If the command COLUMN (30) is issued with the pointer on column 4, then the routine will skip 26 columns and will be in a position to read the 30th column.

(e) ACV:

This routine handles the A-format conversion during run time. It is different from the Fortran A-Conversion routine for the following reasons. In PL 7044 it is assumed that when a variable is outputted in A-format, it is a character variable whose addressing scheme is different from those of ordinary variables. Also while inputting in A-format, initialization of length may have to be done in case the variable is of varying length.

Fig. RN-12 gives a flow chart of the ACV routine. If the character variable is of varying length, the field width w is compared with the maximum length L of the character variable. If $w \geq L$ then the present length l is set to maximum length L, w-L characters are skipped from the input file and finally L characters are read into the L positions of the character variable. If $w < L$ then the present length l is set to the value w and w characters are read into w positions of the variable.

If the character is of non varying type then the field width w is compared with the fixed length l. If $w \geq l$, then w-l characters from the input file are skipped and then l characters are read into the l positions of the variable. If $w < l$ then w characters are read into the variable and the remaining l-w positions are padded with blanks.

In case of an output file, the ACV routine does not have to check whether the variable is of varying or fixed type. It simply compares the present length l or fixed length l as the case may be with w . If $w \geq l$ then w - l blanks are first written on the output file and then l characters from the variable are written out. If $w < l$ then the first w characters of the variable are outputted on the file.

(f) BCV:

This routine is used for bit conversion of bit string x variables. The calling sequence for this routine is,

```
TSX IOHBC.,4
PZE  n
```

where n is the field width. This routine is not available in the Fortran repertoire of conversion routines because bit string variables are not allowed in Fortran. This routine has an action similar to ACV in so far as initialization of present length is concerned. But the bit string occupies only one word whereas a character string may be multiworded. For this reason, the representation is also different in the two cases. Fig. RN-12 which represents the action of ACV may also be taken to be representative of the BCV routine although the address access mechanism is different in the two cases.

7. Error Recovery Routine:

In all cases where an error is detected in a run time I/O routine, control is transferred to this routine with the following calling sequence:

```
TSX  S.EXIT,4
BCI  1,<error code>
```

This routine puts out the error code in a suitable format on the console and the System Output file. It then follows the chain created by the I/O supervisor and flushes the buffers of all the files before handing over the control to the system.

Fig RN-6a
GETITM

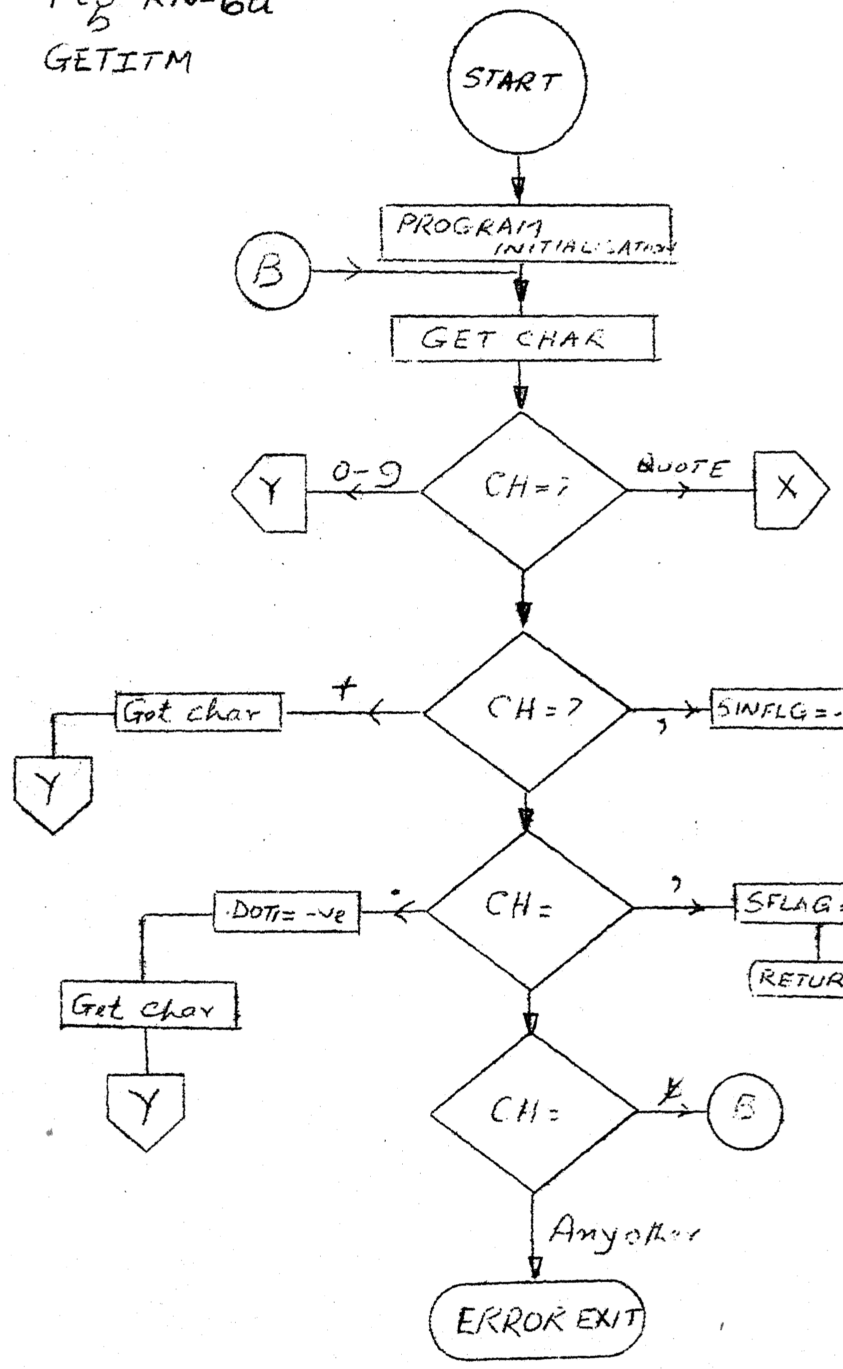


Fig RN·6b
GETITM (CONT)

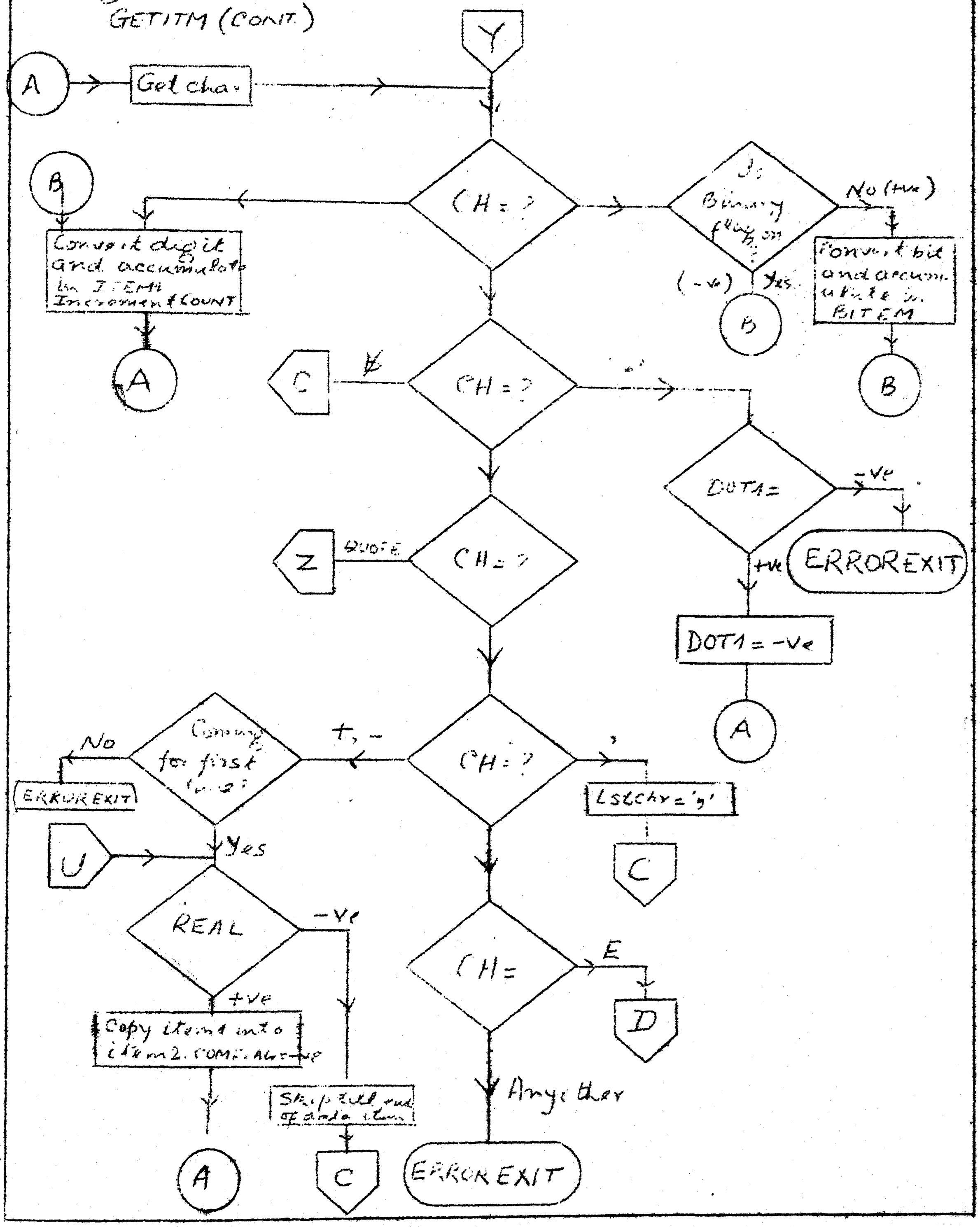
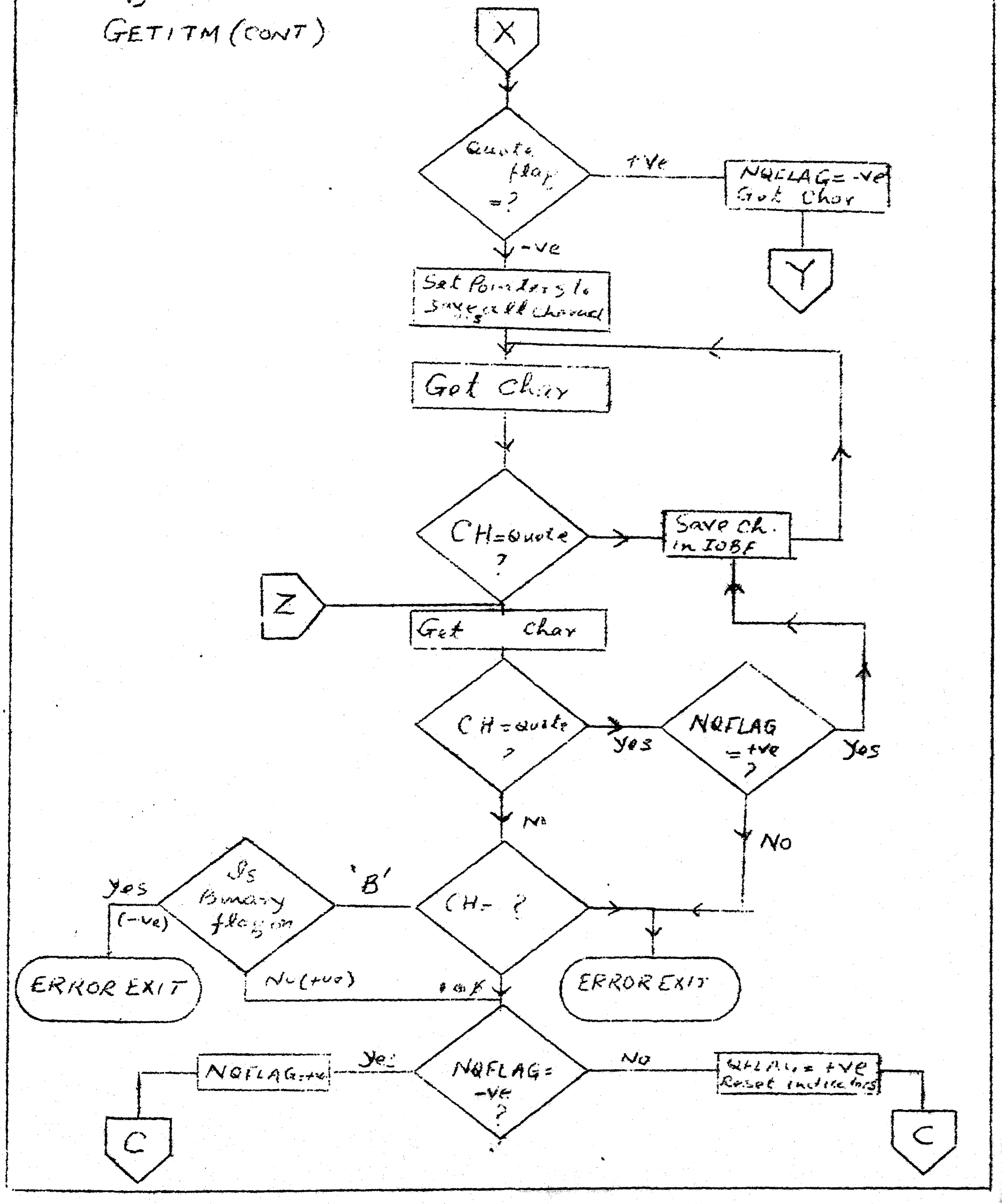


Fig. RN-6c
GETITM (CONT)



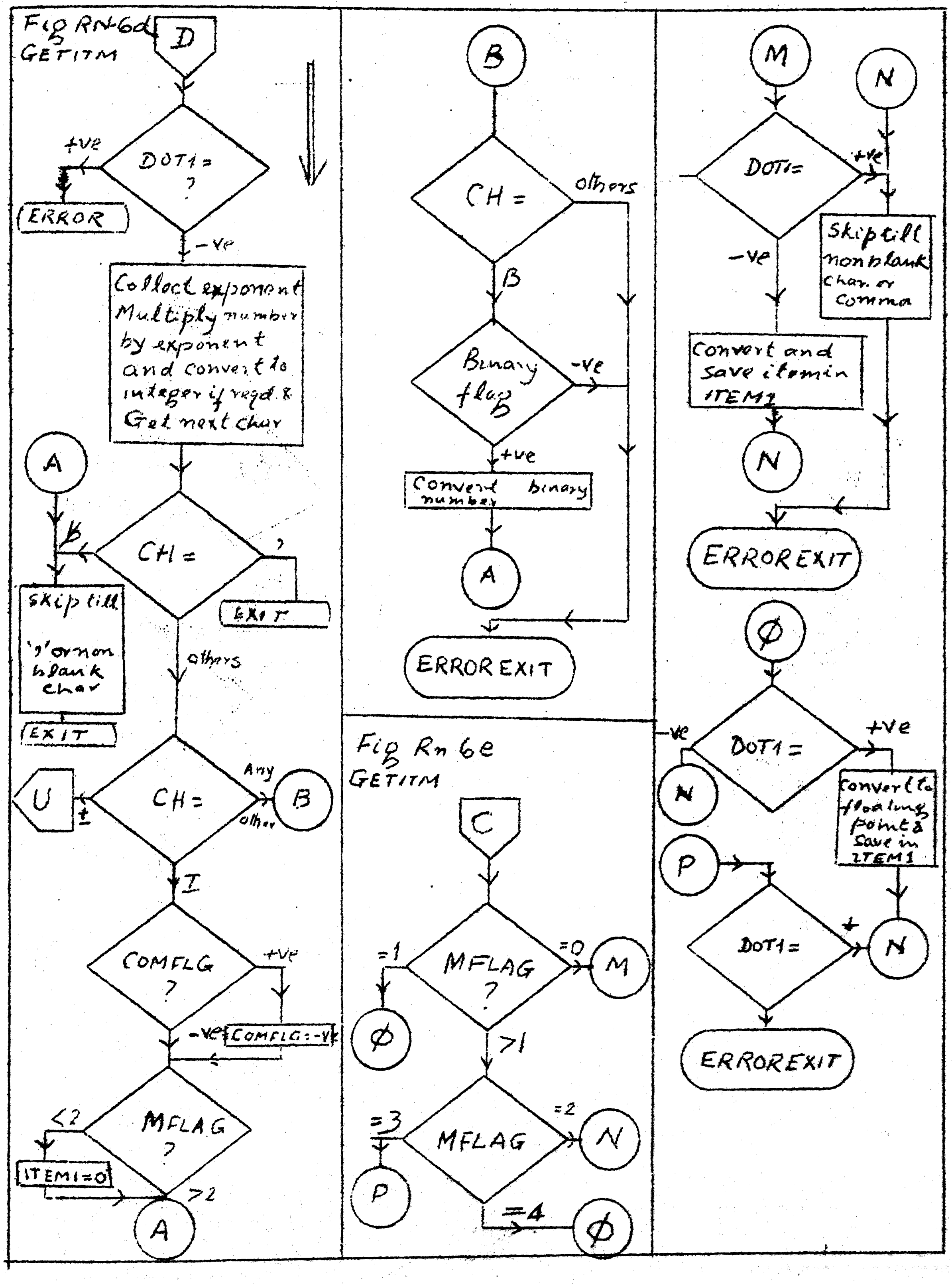
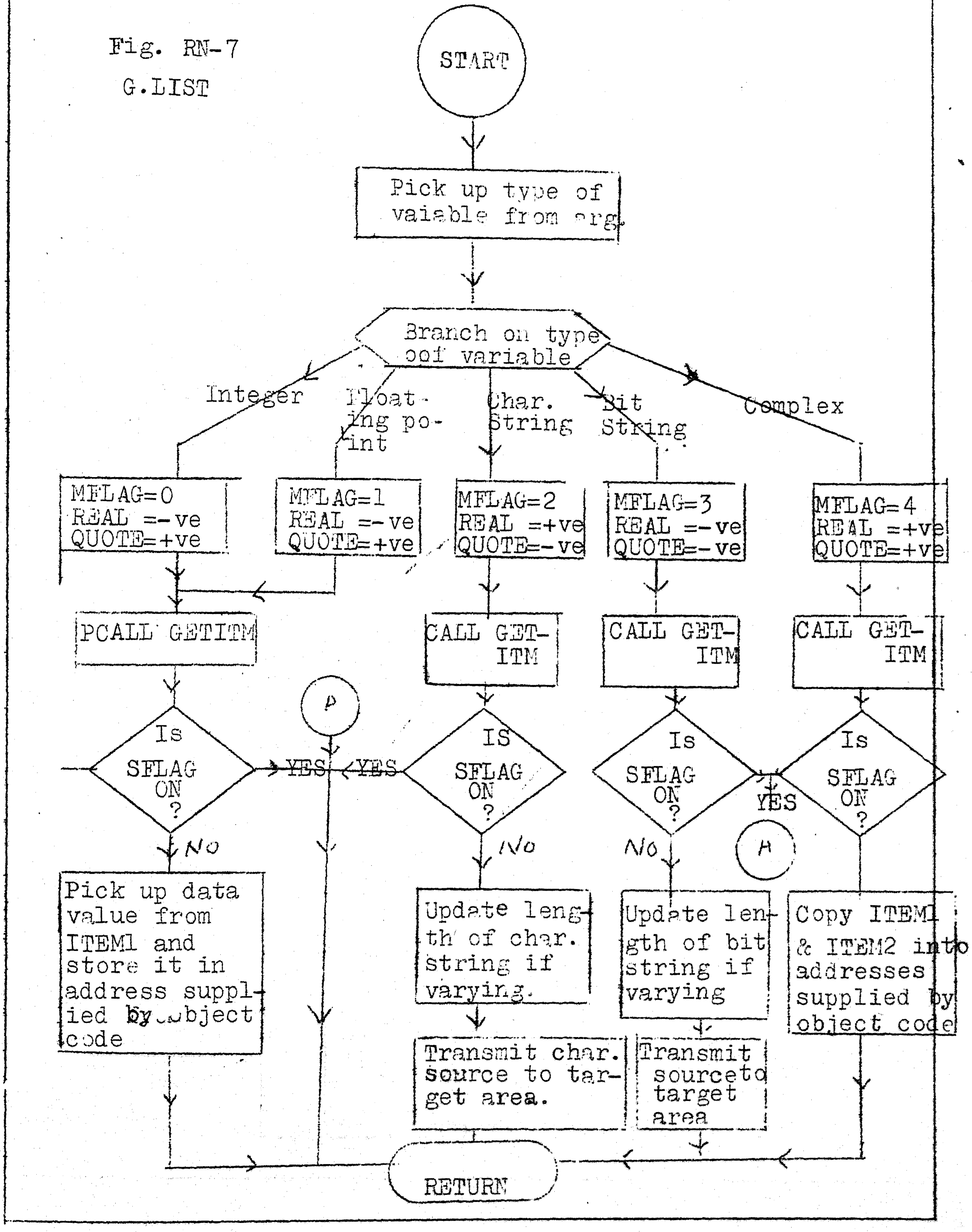


Fig RN-6d GETITM

Fig Rn 6e GETITM

Fig. RN-7
G.LIST


START
Pick up type of vaiable from arg.
Branch on type oof variable
Integer
Floating point
Char. String
Bit String
Complex
MFLAG=0 REAL =-ve QUOTE=+ve
MFLAG=1 REAL =-ve QUOTE=+ve
MFLAG=2 REAL =+ve QUOTE=-ve
MFLAG=3 REAL =-ve QUOTE=-ve
MFLAG=4 REAL =+ve QUOTE=+ve
PCALL GETITM
CALL GET-ITM
CALL GET-ITM
CALL GET-ITM
P
Is SFLAG ON ?
YES
YES
IS SFLAG ON ?
Is SFLAG ON ?
YES
Is SFLAG ON ?
No
No
No
A
Pick up data value from ITEM1 and store it in address supplied by object code
Update length of char. string if varying.
Update length of bit string if varying
Copy ITEM1 & ITEM2 into addresses supplied by object code
Transmit char. source to target area.
Transmit source to target area
RETURN

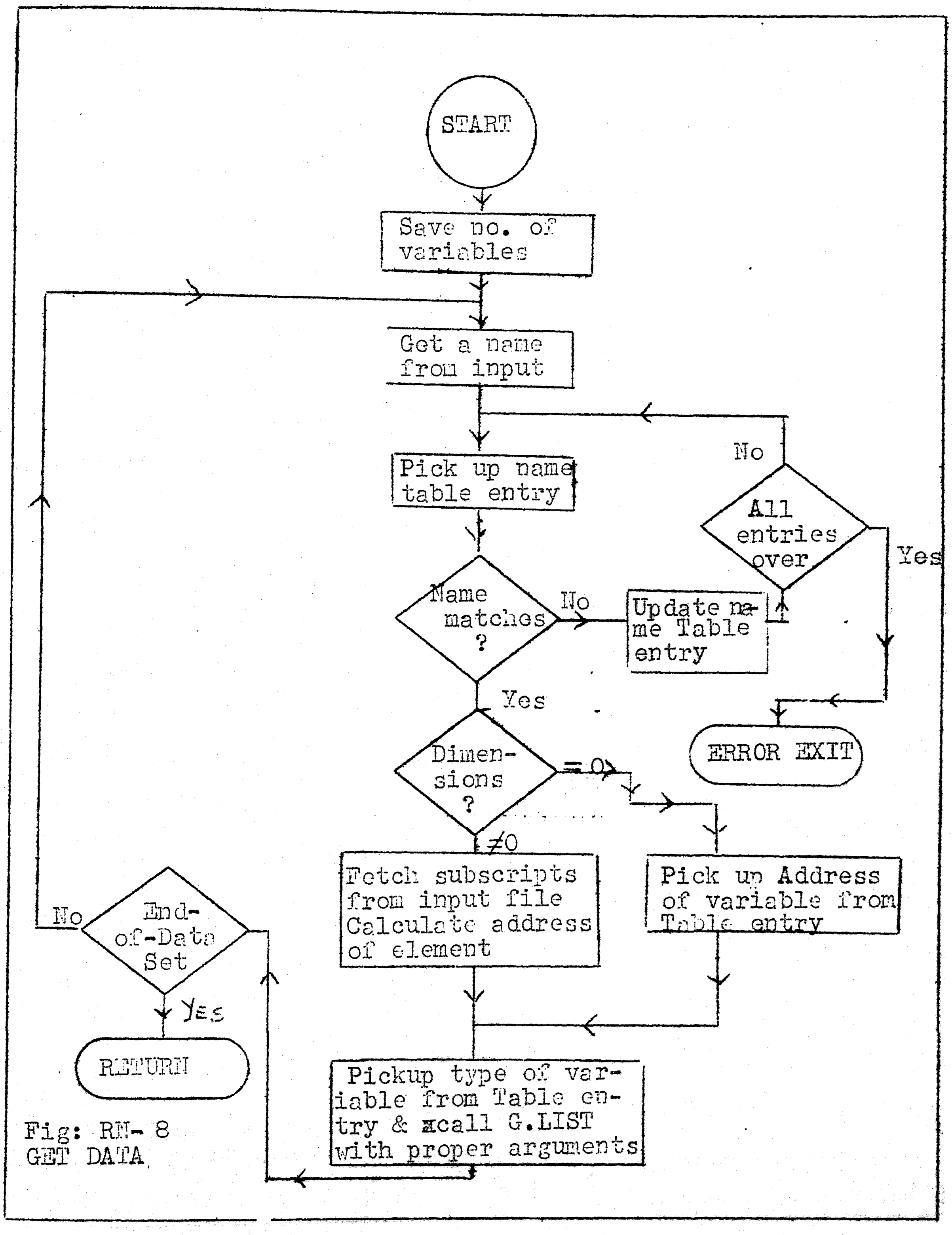
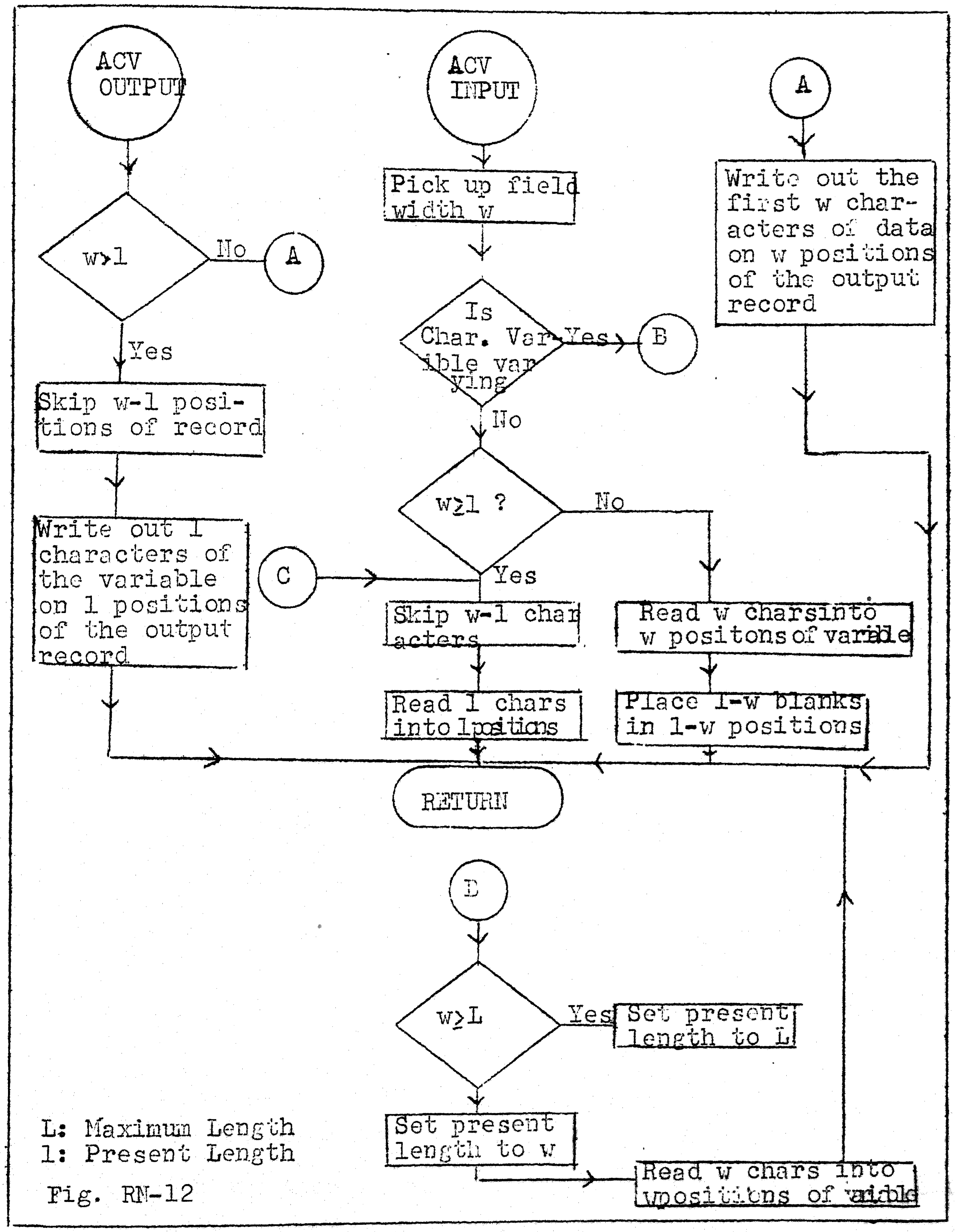


Fig: RN- 8
GET DATA


ACV OUTPUT
w>1
No
A
Yes
Skip w-1 positions of record
Write out l characters of the variable on l positions of the output record
ACV INPUT
Pick up field width w
Is Char. Variable varying
Yes
B
No
w≥l ?
No
C
Yes
Skip w-1 characters
Read l chars into l positions
Read w chars into w positons of variable
Place l-w blanks in l-w positions
A
Write out the first w characters of data on w positions of the output record
RETURN
B
w≥L
Yes
Set present length to L
Set present length to w
Read w chars into w positions of variable

L: Maximum Length
l: Present Length

Fig. RN-12

APPENDIX A

A-1: Character Sets:

For writing the source program in PL-7044, 48-character set is to be used. The 48 - Character set is a subset of the 60 - character set. The 48- character set comprises of 26 alphabets, 10 digits, and 12 special characters.

The special characters are,

Character	Representation
Blank	
Equal sign or assignment symbol	=
Plus sign	+
Minus sign	-
Asterisk or multiply symbol	*
Slash or divide symbol	/
Left parenthesis	(
Right parenthesis	)
Comma	,
Point or period	.
Single quotation mark	'
Dollar symbol	$

To represent the remaining characters of the 60 - character set, which are used in PL/I language, composite symbols are used. The list of composite symbols and the equivalent 60 -xcharacter set representation is given in the following table.

Composite Symbols C Composite Symbols	60-character set Equivalent	Name
..	:	Colon
,.	;	Semicolon
OR	|	"Or" Symbol
AND	&	"AND" Symbol
GT	>	"GREATER THAN" Symbol
LT	<	" LESS THAN" Symbol
NOT	¬	"NOT" Symbol
LE	<=	"LESS THAN EQUAL TO" Symbol
CAT	||	"CONCATENATION" Symbol
**	**	"EXPONENTIATION" Symbol
NL	¬<	"NOT LESS THAN" Symbol
NG	¬>	"NOT GREATER THAN" Symbol
NE	¬=	"NOT EQUAL TO " Symbol
//	%	"PERCENT" Symbol
GE	>=	"GREATER THAN EQUAL TO" Symbol
/*	/*	"START OF COMMENT" Symbol
*/	*/	"END OF COMMENT" Symbol.

With 48- character set, following rules are to be observed:

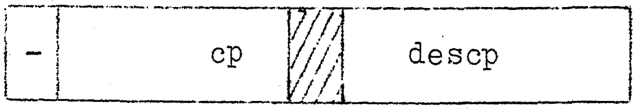
1. The two periods that replace the colon must be immediately preceded by a blank if the preceding character is a period.

2. The two slashes that replace the percent symbol must be immediately preceded by a blank if the preceding character is an asterisk, or immediately followed by a blank if the following character is an asterisk.

3. The sequence " Comma Period " represents a semi colon except when it occurs in a comment or character string.

A2: LEXICAL REPRESENTATION OF VARIOUS LEXUNITS:

Operators



where,

"cp" is compare priority, and

"descp" is the description number which uniquely identifies an operator and is used in table look up etc.

"descp" and " cp " for various operators are as follows:

descp	cp	Operator
1		Unary minus (assigned by the expression processor)
2	11	NOT
3	11	**
4	10	*
5	10	/
6	9	+
7	9	-
8	7	CAT
9		Relational '=' (EQ)(Assigned by the expression processor)
10	6	NE
11	6	GT
12	6	GE
13	6	NG
14	6	LT
15	6	LE
16	6	NL

Contd.

17	5	AND
18	4	OR
19	4	,
20	3	)
21	11	(
22	11	=
23	2	;.
24	4	..
25	–	.

<u>Operands</u>:

Unlike operators, operands contain both the type information and their description. Type is available in a word called TYPE1, whereas the description is available in other associated words. The actual operand is collected in a six word buffer called LEXBUF.

$\boxed{\qquad\qquad\qquad \text{type}}$ TYPE1

where " type " is

<u>type</u>	<u>Operand</u>
1	Pseudo reserved word or Key word
2	Identifier or non Key word
3	Integer Constant
4	Floating point constant
5	Bit string constant
6	Character string constant

Pseudo Reserved Words:

These represent identifiers having special meaning. The actual name string is available in words starting with LEXBUF and word count and character count is available in WCOUNT and CCOUNT respectively. The code number to identify a particular pseudo reserved word is available in CODE.

Identifiers:

These are symbols which have no special meaning. Their description is exactly like pseudo reserved words except that they have no code numbers.

Integer/Floating point constant:

Actual constant, converted from BCD to Binary, is available in LEXBUF.

Actual bit string, left justified, is available in LEXBUF and bit count is available in CCOUNT.

Character String Constant:

Actual character string is available in words starting with LEXBUF, and word count and character count is available in WCOUNT and CCOUNT respectively.

APPENDIX B

<u>Constants in PL-7044</u>:

Constants that can be used in PL 7044 are of the following five types:

1. Arithmetic constants,
2. String constants,
3. Statement label constant,
4. Entry constants, and
5. File constants.

<u>Arithmetic Constants</u>:

Arithmetic constants can be one of the three types.

a) Integer Constants:

Maximum length of integer constants allowed in 27 bits i.e. an integer greater than $2^{27}-1$ (163708927) can be used.
Examples: 121, 0, 7777 etc.

b) Floating Point Constants:

Floating point constant of PL 7044 is exactly like the floating point constant of IBM 7044 - FORTRAN i.e. the range is 10^{-38} to 10^{+38} with 8 decimal digit precision.

Examples: 1.23, 1.0 E-10, .0003, 5E2 etc.

c) Complex Constants:

Complex constants can be of two types

i) purely imaginary constant e.g. $\pm$ 2I and

ii) Complex constants having real and imaginary part both, e.g. 1 $\pm$ 2I.

Complex constant is stored as real constant and imaginary constant. Both these constants are floating point constants. The internal representation of the two constants shown here would be 0 ± 2.0I and 1.0 ± 2.0I respectively.

String Constants:

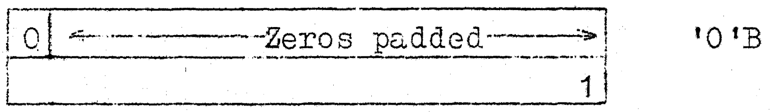
String constants are of two types:

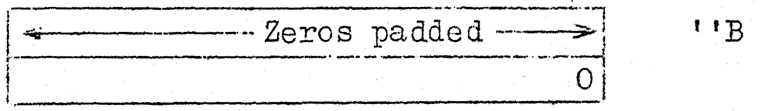
a) Bit string constants:

Each digit can be either 1 or 0 only. Its representation is as follows -

'101'BB, '0'B , ''B

The bit strings '0'B and ''B are different, first being a bit string of length 1 whereas the latter is a null bit string. Maximum length of a bit string can be 36 (word size of IBM 7044). Bit string is stored left justified with remaining bit positions filled with zeros. Each constant at run time would be stored in two consecutive words, first word containing the actual bit string and the second word contains the bit string length. The representation of '0'B and ''B would be as follows





b) Character String Constants:

A character string can be constructed using any of the 48 characters of the 48-character set. The representation of a character string is as follows:

'GARIBI HATAO'

'C.I.=PR*(1.+RT/100.)**NY-PR'

' '

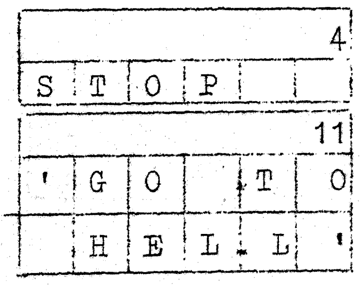
''

The last two character strings are not same. First of the two is a character string of length 1 containing one 'blank' character. The last one is a null character string i.e. string of length zero. The length of character string represents the number of characters in the character string. Maximum length of a character string constant can only be 30.

If quote happens to be one of the characters of the string then it should be repeated i.e. two quotes for each quote that is a part of the character string e.g.

TEXT	REPRESENTATION
IT WON'T HAPPEN	'IT WON"T HAPPEN'
" DON'T GO", SHE SAID	""""DON"T GO,"""" SHE SAID'
"	"""

Each character string constant is stored in consecutive words with remaining right most character positions of the last word, if any, padded with blanks (Ø60 in IBM 7044). The first word in the storage for a character string constant, stores its length.



'STOP'

'GO TO HELL'

Repetition Factor:

The representation of a long string containing a repetitive substring can be made quite concise by using 'repetition factor' e.g.

Expanded form	Short form
'1111111'B	(7)'1'B
'ABCABCABC'	(3)'ABC'
'10101010'B	(4)'10'B

The length restriction of 30 characters holds for the expanded string constant. Thus expanded bit string constant should not use more than 36 bits while the expanded character string constant should not be more than 30 characters long.

Statement Label Constants:

Statement label constants are the symbols prefixed to any statement except the PROCEDURE and ENTRY statements. e.g. consider the following statements -

```
COMPUTE:  A = B+C;
BRANCH:  GO TO HELL;
LAB1:LAB2:LAB3:DO;
```

In the above example COMPUTE, BRANCH, LAB1,LAB2,LAB3 are statement label constants. The constructed label constant format is as follows

bl. no.	Do-number	▨	Label Address

where,

" blno." is the serial number of the block to which the statement appended to the label constants, is is internal.

" Do-number" is the serial number of the immediately encompassing do-group. In case no such do-group exists, it is zero.

" Label Address" is the starting address of the code generated for that statement.

This constructed label constant is used in label assignment.

Entry Constants;

Entry constants are symbols prefixed to the PROCEDURE and ENTRY statement. In the following example FACTORIAL, MAIN, INTEGER, COMPLEX and FLOATINGPOINT are examples of entry constants.

```
FACTORIAL:  PROCEDURE;
MAIN:  PROCEDURE OPTIONS (MAIN);
COMPLEX: FLOATINGPOINT:INTEGER:
                              PROCEDURE(X) RETURNS (REAL(10,0));
```

Since entry variables are not allowed in PL 7044, constructed entry constant is not defined. However since entry constants can be passed as arguments, a procedure-header-word is defined with every entry constant, whose format is as shown below,

x	prglst address	y	text address

where,

" yx" is the serial number of the block to which the code belongs.

" prglst address" is the address of the prologue list
generated, if any, for the entry point, otherwise,
it is zero.

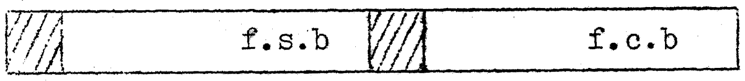
" text address" is the address where the control should be
transferred at the time of invocation.

<u>File Constants</u>:

File constants are names given to files which are then referred to by these names, e.g.

```
DECLARE INPUT FILE;
OPEN INPUT;
```

INPUT is a file constant here, constructed file constant representation while transmitting as an argument is,



where,

f.c.b. is file control block address and

f.s.b. is file status block address.

APPENDIX C

Names in PL 7044:

Every variable defined in PL/I is given an alphameric name so that a reference can be made to it by this name. Names can be of two types:

i) Non qualified or Simple names

ii) Qualified names

Non Qualified Names (NQ-name):

Following rules should be observed while constructing a NQ-name:

1i) first character (or the only character) of a name should be one of the 26 alphabets.

ii) Remaining characters can be any of the 26 alphabets and 10 digits.

iii) More than 30 characters should not be used in forming a name.

(iv) There should not be any intervening blanks since blank serves to delimit a name.

Valid names:

BOND007, APOLLO, I101, J

Invalid names:

1A, BOND.1, RAM LAL

Qualified names (Q-name):

A Qualified name can be defined as

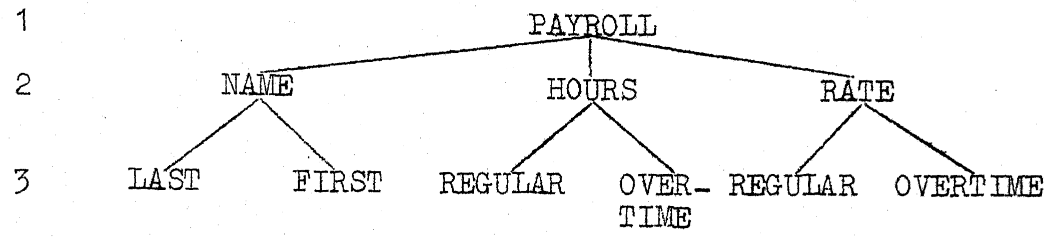
NQ-name $\{$.NQ-name$\}$ ...

A qualified name is used to refer to minor structures or base elements of a structure.

Consider the following structure:

```
DECLARE 1 PAYROLL,
          2 NAME,
            3 LAST,
            3 FIRST,
          2 HOURS,
            3 REGULAR,
            3 OVERTIME,
          2 RATE,
            3 REGULAR,
            3 OVERTIME;
```

The tree corresponding to this structure would be



Qualification in the Q-name is in the order of levels; that is the name at the highest level must appear first, with the name at the deepest level appearing last.

Since any of the names in a structure, except the major structure name itself, need not be unique, Q-name makes it unique. Thus in the above example only REGULAR is ambiguous but PAYROLL. HOURS.REGULAR and PAYROLL.RATE.REGULAR are not. However, names of all the nodes from root to the node in question, need not be used in qualification, if it does not give rise to ambiguity. Though LAST and FIRST uniquely represent the concerned nodes, PAYROLL.NAME.LAST and PAYROLL.NAME.FIRST represent the complete qualifications.

APPENDIX D

D-1: Variable Types in PL-7044:

Following types of variables are provided in PL 7044,

1. Arithmetic variables
2. String variables
   a) Bit string variables
   b) Character string variables
3. Label variables

Notation: 'varadr' represents the address of the first order storage assigned to the variable, and is available in the symbol table.

Arithmetic Variables:

Arithmetic variables are names of arithmetic data items. These names have been given the characteristics (i.e. attributes) of Base, Scale, Mode and Precision. PL 7044 has only BINARY base though it accepts DECIMAL, if specified yet BINARY is implied. This is because IBM 7044 has no DECIMAL arithmetic hardware. Similarly PL 7044 accepts both FIXED point and FLOATing point scales, with precision specified, for the sake of compatability but internally it works with only two representations - Floating point (8 bit characteristic and 2 bit mantissa in normalised representation) or Integer (27 bit).

A specification of type FIXED (m,0) or FIXED (m) implies integer representation in PL 7044 whereas FIXED (m,n),n≠0 and FLOAT (m,n) implies floating point representation.

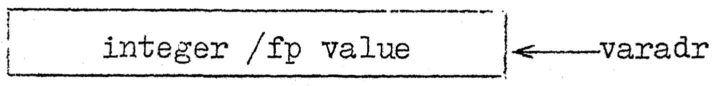
Output from PL 7044 compiler is only in these modes though input is accepted in the general format, again for compatability sake.

Thus arithmetic variables can be of three types -

1. Integer, 2, Floating point, and 3. Complex.

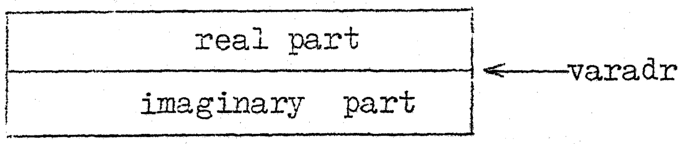
Integer and Floating Point Variables:

Each integer or floating point word requires one storage word. In case of formal parameters of these types, this word points to the parent integer or floating point variable passed as the argument i.e. one level of indirection is involved.



Complex Variables:

Each complex variable is assigned two consecutive words in the storage. In case of a complex formal parameter, both these words point to the corresponding words of the argument passed. Thus both real and imaginary parts involve one level of indirection.



String Variables:

String variables are names of string data items. These names have been given string attributes. The general format of defining string variables is, name name $\left\{\begin{array}{l}\text{BIT}\\ \text{CHARACTER}\end{array}\right\}$ [(length)] [VARYING]

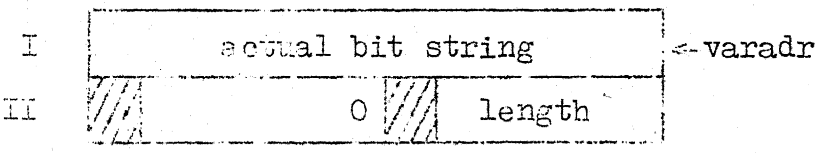
The length attribute specifies the length of a fixed length string or the maximum length of a variable length string. When the length attribute is omitted a length of 1 is assumed.

While there is no restriction on the length of a character string variable, a bit string variable length can not be more than 36 (IBM-7044 word size).

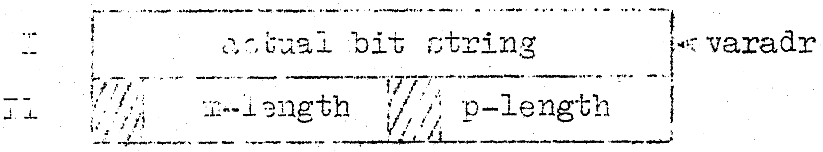
The VARYING attribute specifies that the variable is to represent varying length strings. The current length at any time is the length of the current value.

BIT STRING:

Like complex variable, a bit string variable also requires two consecutive storage words, The format is



Fixed Length Bit String.



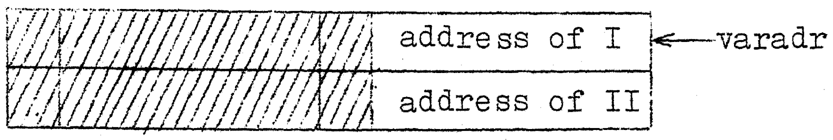
Varying Length Bit String.

where,

"m-length" is maximum length, and

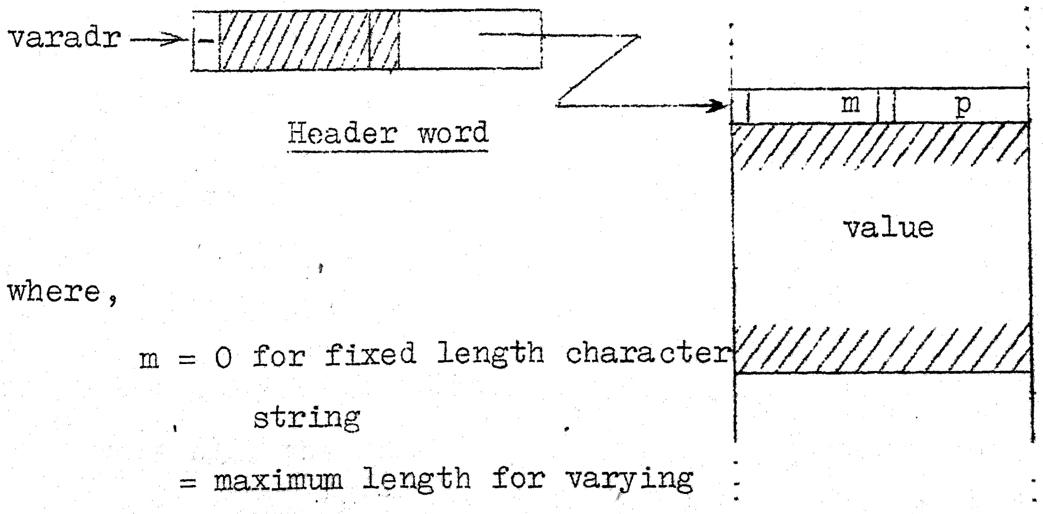
"p-length" is present length.

For a formal parameter, the representation is exactly like that a formal complex parameter



CHARACTER STRING:

Unlike arithmetic variable and bit string variable, character string variable is of non standard size, since actual size depends on the declared length of the variable. Therefore storage for a character string is assigned in two parts. The actual area for holding the value of the variable is given from the second order storage (i.e. run-time-stack). However, one word is assigned from the first order storage which points to the actual area. This word is called a header-word and its address is available in the symbol table. The representation is as follows:



Header word

Run time stack

where,

m = 0 for fixed length character string

= maximum length for varying character string

p = present length

The reason for making the sign of the header word negative would be explained shortly.

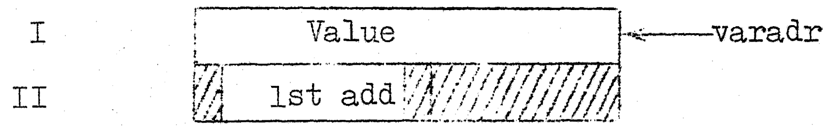
For a formal parameter of character string variable type, no special treatment is given, since the header word of the argument is copied in the parameter storage word and then it becomes exactly like that of a normal variable.

Label Variable:

A label variable is a variable that has as values, statement label constants (slc). The attribute specification may include the values that the name can have during execution of the program. General format is,

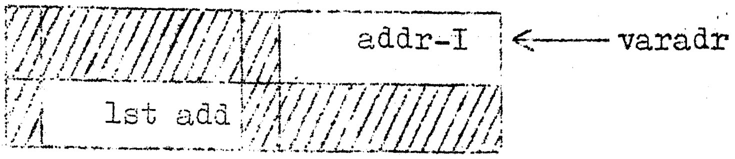
LABEL [(slc [, slc]...)]

Label variable is given two words from the first order storage area. The representation is as follows,



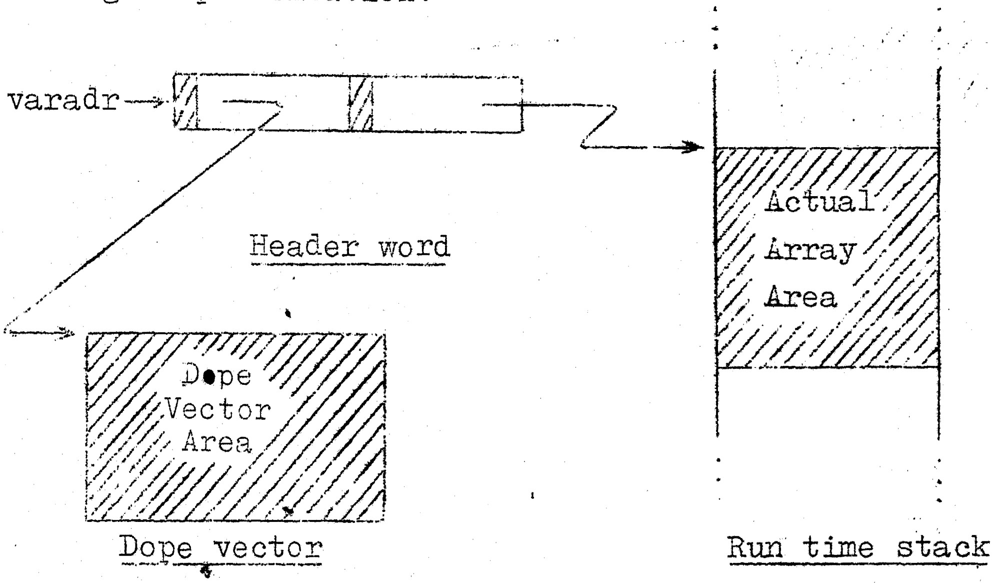
where " 1st add" is the address of the label-list, if one specified otherwise it is zero. Label-list is an array of constructed-label-constants generated from the list specified after the keyword LABEL.

For a formal parameter which is given the attributes of label variable, the first word points to the address of the first word of the argument passed whereas the second word contains '1st add' of the argument.



Till now our discussion was limited to scalar variables. We will now consider Arrays.

Storage representation:



(3n+1 words where n is the number of dimensions.)

Thus it can be seen that array consists of three disjoint sets of storage locations. Every array is allocated a header from the first order storage (one word for all types of arrays except for label arrays with list which have two word header. Second word stores the list address), whose address is filled in the symbol table. Header word points to the dope vector and the actual area. Dope vector consists of information about bounds etc. and is assigned area from the first order storage area (dope vector region). The actual area for the elements of the array is assigned from the second order storage or the run time stack.

There is no difference between the representation of a formal array and that of a normal array.

The format of array elements is exactly like the format of scalar variables for all types; except for the character variable; for an element of a character array, there is no header word. Therefore, if an element is transmitted as an argument, the address supplied would directly point to the character element whereas for the character scalar, it would point to the header word and hence one level of indirection would have to be done to get the address of the actual character string element. To distinguish between these two cases the sign of header for a character scalar is made negative.

D-2 Major Types and Minor Types:

All the symbols which can be given an attribute in PL 7044, have been classified under two main categories:

1. Major Types
2. Minor Types

Major Types:

Following is the list of major types :

Major Type	Description
1	Normal Array
2	Formal Array
3	Static Array
4	Not applicable (Defined Array)
5	Normal Scalar
6	Formal Scalar
7	Static Scalar
8	Not applicable (Defined Scalar)
9	PROCEDURE without entry list
10	PROCEDURE with entry list
11	PROCEDURE BUILTIN
12	PROCEDURE MAP
13	PROCEDURE GENERIC
14	PROCEDURE Open ended
15	Pseudo Variable
16	Normal major/minor structure
17	Formal major/minor structure
18	Label constants
19	Formal entry without entry list
20	Formal entry with entry list
21	File constant
22	Formal file
23	PROCEDURE MAP with entry list
24	PROCEDURE BUILTIN where arrays are allowed.

The variables, corresponding to major types 1 to 8 are further divided into minor types. The list of minor types is as follows:

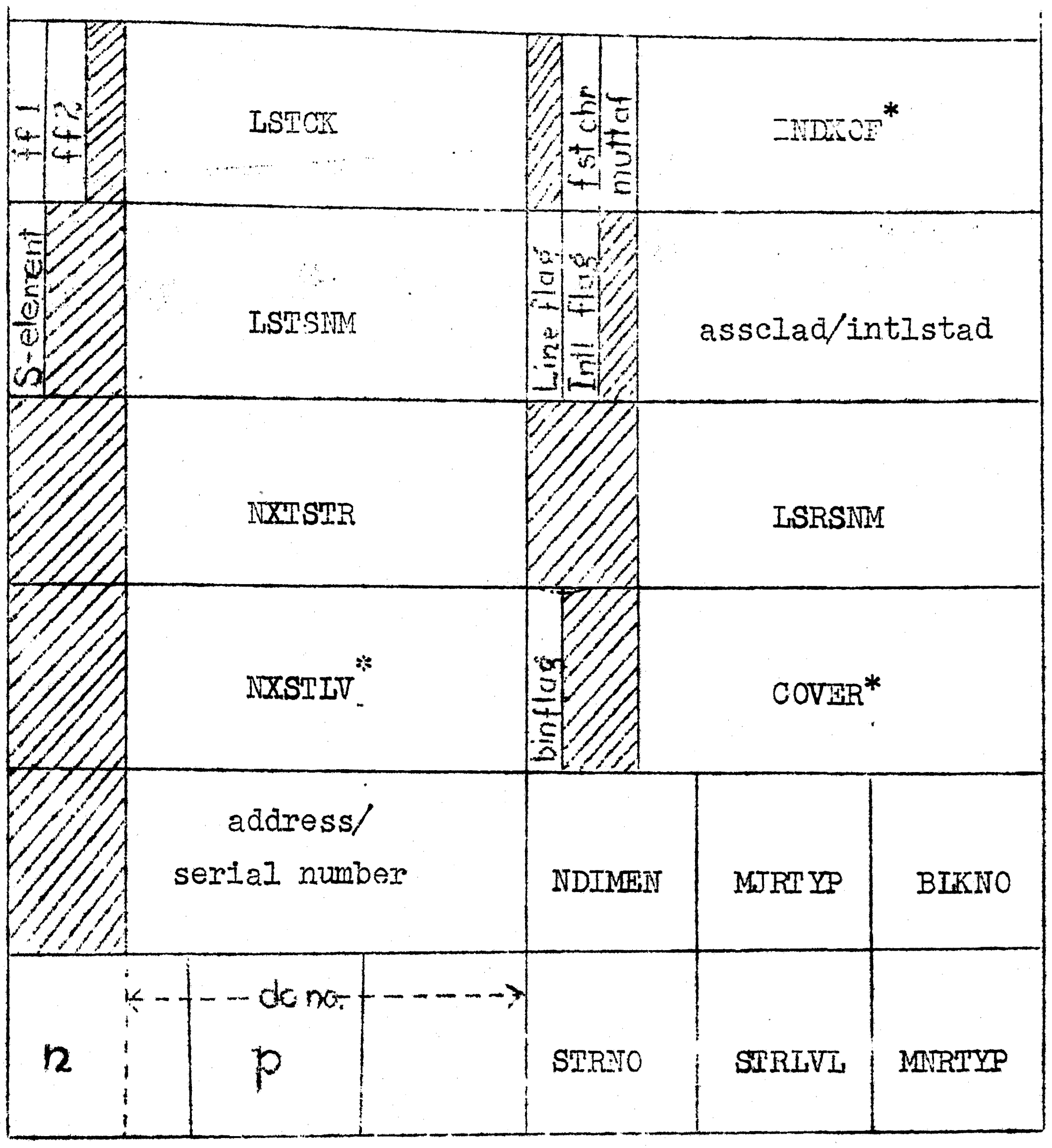
Minor Type	Description
1	Integer
2	Floating point
3	Complex
4	Not applicable (integer numeric picture)
5	Not applicable (floating point numeric picture)
6	Bit string- Fixed length
7	Bit string - varying length
8	Character string - fixed length
9	Character string - varying length
10	Not applicable (non numeric picture)
11	Label without label list
12	Label with label list

D-3: <u>Symbol Table Representation of Various Major Types</u>:

Before describing about individual major types, it would be appropriate to look at the typical symbol table cell shown on the next page.

<u>Nomenclature</u>:

LSTCK:	Address of the last cell in stack Index table offset.
INDXOF:	Index table offset.
LSTSNM:	Address of the last cell in stack with same ~~x~~name



SYMBOL TABLE CELL

Note: Items with asterisk are stored in 2's complement notation.

NXTSTR:	Address of the cell next in structure.
LSRSNM:	Address of the last cell in stack with same name in the structure.
NXSTLV:	Address of the cell representing next sister node.
COVER:	Address of the covering item cell.
NDIMEN:	Number of dimensions for variables or number of parameters for an entry name, zero if not applicable.
MJRTYP:	Major type
BLKNO:	Block number
STRNO:	Structure number
STRLVL:	Structure level
MNRTYP:	Minor type (for entry constants it stores the minor type of result returned).
address/:	Address of first order storage for variables.
serial no	Address of PHW for entry constants. Label serial number for label constants. File serial number for file constant.
assclad :	Associated cell address for built in procedures, and procedures with entry list.
intlstad:	Initial list address for structure elements
n,p:	Precision information for arithmetic variables. Primary and secondary types of file for a file constant
dono.:	DO number for a label constant.

<u>Flags</u>:

ff1:	Formal flag -1, '1' if explicit declaration in DECLARE statement occurs, '0' otherwise.

ff2:	Formal flag-2, '1' for formal parameter
fstchr:	first character flag, '1' if initial character of the name is I-N, '0' otherwise.
multdf:	multiple definition flag, set to '1' if a declaration at level 0 or 1 already exists in ablock and another declaration is tried in the same block having same name.
likeflag:	'1' if this structure is expanded by LIKE attribute
Intlflag:	Initial flag, '1' if this variable has been initialised by INITIAL attribute in DECLARE statement.
s-element:	Structure element flag, '1' if it is a base element of a structure
bin flag:	Binary flag, '1' if the arithmetic variable is given a base of BINARY.

Note: Fields and flags which are irrelevant to a major type, contain zeros.

Symbol table representation of variables, file constants and label constants has nothing extra than what is shown in the symbol table cell.

Symbol table representation of entry constants having entry list and generic-procedure would be described below because they involve additional cells. Format for a builtin procedure is described in Appendix H.

Symbol table representation of entry constants having entry list:

For an entry constant having entry list, there are two cells-main and associated, in the symbol table. The main cell is as usual and points to the associated cell (assclad). The associated cell is used to store the type information of various arguments.

Format of associated cell:

NARG	▨	▨
ARG-1	ARG-2	ARG-3
ARG-4	ARG-5	ARG-6
ARG-7	ARG-8	ARG-9
ARG-10	ARG-11	ARG-12
ARG-13	ARG-14	ARG-15

where NARG is number of arguments in the entry list. For the n-th parameter, ARG-n field consists of

pmy-type	sndy-type

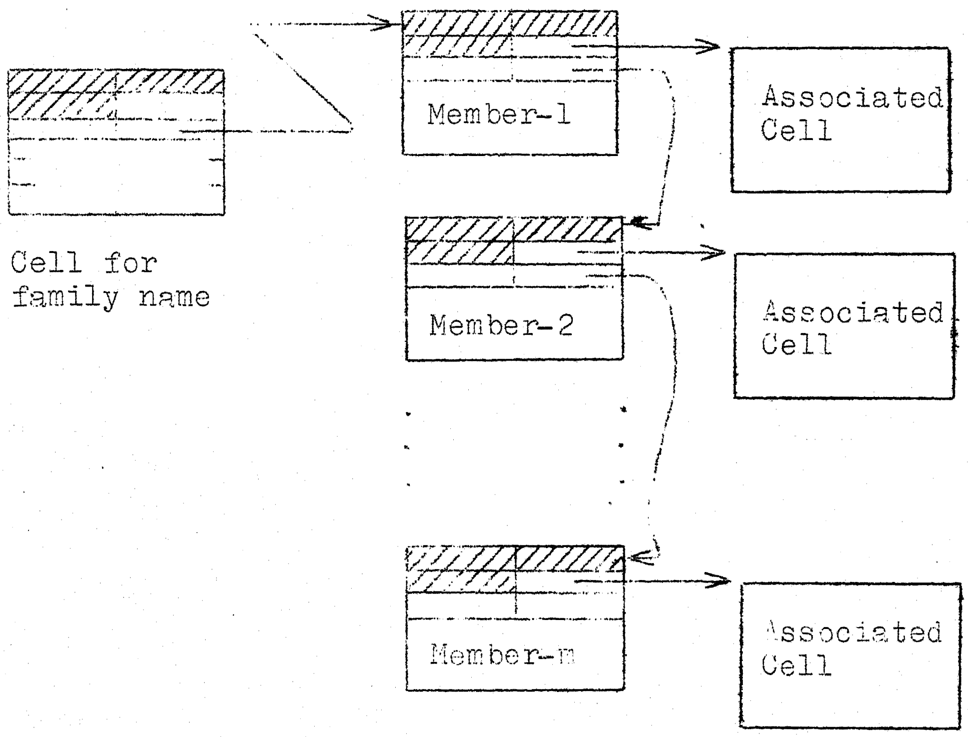
where " pmy-type " is one of the following:

pmy-type		Parameter descriptor
0	:	Star/Null
i,(i=1-5)	:	i-dimensional array
6	:	Structure
7	:	Scalar
8	:	Entry
9	:	File

" sndy-type " is one of the 12 minor types described earlier. For an n-argument entry list, remaining 15-n (n $\leq$ 15) ARG-fields would contain zero.

Because of the limitation put by the size of the cell, more than 15 parameters can not be specified in an entry list.

<u>Symbol table representation for GENERIC family</u>:



(The hatched portion shows the usual pointers).

Each member of the family is represented exactly like an entry constant with entry list. However, a chain is formed starting from the GENERIC cell which links all the members. LSRSNM is used to store this pointer. The chain is terminated by '0' in neat member address.

APPENDIX E

A program organised to explain the concepts of scope of declaration, extension of scope and use of BUILTIN attribute:

```
EXT1:   PROCEDURE OPTIONS (MAIN);
        L1: DECLARE ((A,B) EXTERNAL, C,SQRT(10)) FIXED (6,3),
        S1: GET LIST (A,B, SQRT);
        L2: CALL INT1;
        L3: B=EXT2(SQRT(A));
        L4: CALL INT2;
        S2: Statement
        INT1: PROCEDURE;
              L5: DECLARE (B,C(10)) FIXED (6,3), D STATIC;
              S3: C = B+D+2*SQRT+E;
                  END INT1;
        INT2: PROCEDURE;
              L6: DECLARE C FIXED (6,3), A FLOAT(8),SQRT BUILTIN;
              S4: C=B+0.7*SQRT(C);
                  END INT2;
              S5: Statement
        END EXT1;
EXT2:   PROCEDURE (X) RECURSIVE;
        L7: DECLARE (A,B) FIXED (6,3) EXTERNAL,
                    C FLOAT (4) INITIAL (0), X FIXED (5,0);
        S6: IF X-2* X/2=0 THEN CALL INT3;
        S7: GET LIST (B);
        INT3: PROCEDURE;
              L8: DECLARE (A) FLOAT (4);
              S8: C = B+A
                  END INT3;
        S9: IF X=1 THEN RETURN (1.);
                    ELSE RETURN (C+X*EXT2 (X-1));
            END EXT2;
```

Note: " Statement" refers to a PL-7044 statement.

A table would be made to show the attributes and scopes of various variables.

Statement Label	Name	Attributes	Scope
EXT1	EXT1	ENTRY,EXTERNAL	Entire program
L1	A	FIXED,EXTERNAL	All of EXT1 except INT2 and All of EXT2 except INT3
L1	B	FIXED,EXTERNAL	All of EXT1 except INT1 and All of EXT2
L1	C	FIXED,INTERNAL	All of EXT1 except INT1 and INT2
L1	SQRT	FIXED,INTERNAL, Array	All of EXT1 except INT2
INT1	INT1	ENTRY,INTERNAL	All of EXT1
L5	B	FIXED, INTERNAL	All of INT1
L5	C	FIXED,INTERNAL, Array	All of INT1
S3	E	Default	All of EXT1
INT2	INT2	ENTRY,INTERNAL	All of EXT1
L6	C	FIXED,INTERNAL	All of INT2
L6	A	FLOAT,INTERNAL	All of INT2
L7	SQRT	BUILTIN	All of INT2
EXT2	EXT2	ENTRY,EXTERNAL	Entire program
L7	A,B	FIXED,EXTERNAL	Same A and B as declared in L1
L7	C	FLOAT,INTERNAL	All of EXT2
L7	X	FIXED, INTERNAL (parameter)	All of EXT2
INT3	INT3	ENTRY,INTERNAL	All of EXT2
L8	A	FLOAT,INTERNAL	All of INT3

Examination of this program will reveal the following facts:

i) There are three separate locations identified by same name, A. The first of these is allocated by the declaration in statement L1 as a fixed point decimal number with the EXTERNAL attribute. As such it is also given the STATIC attribute by default. The second location name 'A',declared at L6, refers to a 8 digit floating point decimal number known to only procedure INT2.
The third is a floating point decimal number with the internal attribute, declared at L8, which remains allocated during the invocation of procedure INT3.
The A declared in L7 is the same as A declared in L1.

ii) There are two different locations with name B. The first is declared in statement L1, whereas the second is declared in statement L5 and is known only to procedure INT1.
Though both the B's are fixed point decimal numbers with same precision, the fact that second B is declared, explicitly, in an internal procedure results in a separate allocation and release upon termination of that procedure.

iii) The program assigns four different locations using the name C, one of them being the header of an array.

iv) The name SQRT is having two different meanings in this program. SQRT is declared in L1 as a one dimensional array of floating point decimal numbers with single precision. It's use in statement L3 is as a floating point variable.

In statement S3 it is being used in an aggregate expression. The use of SQRT in statement S4 is as a built in procedure and it represents a call to the builtin procedure with name SQRT. Thus use of SQRT in external procedure L1 as an array does not bar its use in an internal procedure as a builtin name, if declared appropriately.

v) The procedure EXT2 has been given RECURSIVE attribute and it is being called from within itself at statement S9. In such cases a rule for terminating the recursion is also to be given, otherwise it will form an infinite loop.
At each call of EXT2, a value of B is read in from input media but only the value which is existing when a call is made to INT3, will be transmitted to INT3.

APPENDIX F

First Order Storage Assignment For Various Blocks of a Program:

To illustrate the first order storage assignment for various blocks of a program, program of Appendix E would be considered once again.

a. Block Predecessor Table:

BLPDTB (at compile time)	b.no.	name	Run time coding
			BLPDTB EQU *
0	0	imaginary	OCT 0
0	1	EXT1	OCT 0
1	2	INT1	OCT 1
1	3	INT2	OCT 1
0	4	EXT2	OCT 0
4	5	INT3	OCT 4

b. Block Storage Table:

		STGTAB EQU *
0	imaginary	PZE BE.00,,BS.00
1	EXT1	PZE BE.01,,BS.01
2	INT1	PZE BE.02,,BS.02
3	INT2	PZE BE.03,,BS.03
4	EXT2	PZE BE.04,,BS.04
5	INT3	PZE BE.05,,BS.05

C. Labels Generated:

```
AS.00   EQU   BS.00+≠≠SEA(00)
TS.00   EQU   AS.00+≠≠DPVC(00)
BE.00   EQU   TS.00+≠≠TA(00)

BS.01   EQU   BE.00
AS.01   EQU   BS.01+≠≠SEA(01)
TS.01   EQU   AS.01+≠≠DPVC(01)
BE.01   EQU   TS.01+≠≠TA(01)

BS.02   EQU   BE.01
AS.02   EQU   BS.02+≠≠SEA(02)
TS.02   EQU   AS.02+≠≠DPVC(02)
BE.02   EQU   TS.02+≠≠TA(02)

BS.03   EQU   BE.01
AS.03   EQU   BS.03+≠≠SEA(03)
TS.03   EQU   AS.03+≠≠DPVC(03)
BE.03   EQU   TS.03+≠≠TA(03)

BS.04   EQU   BE.00
AS.04   EQU   BS.04+≠≠SEA(04)
TS.04   EQU   AS.04+≠≠DPVC(04)
BE.04   EQU   TS.04+≠≠TA(04)

BS.05   EQU   BE.04
AS.05   EQU   BS.05+≠≠SEA(05)
TS.05   EQU   AS.05+≠≠DPVC(05)
BE.05   EQU   TS.05+≠≠TA(05)

BS.00   EQU   *
        BSS   MXFSAR
```

where,

MXFSAR represents maximum first order storage area required

SEA(bn) represents the scalar elements area for block 'bn'

DPVC(bn)represents the dope rector area for block 'bn'

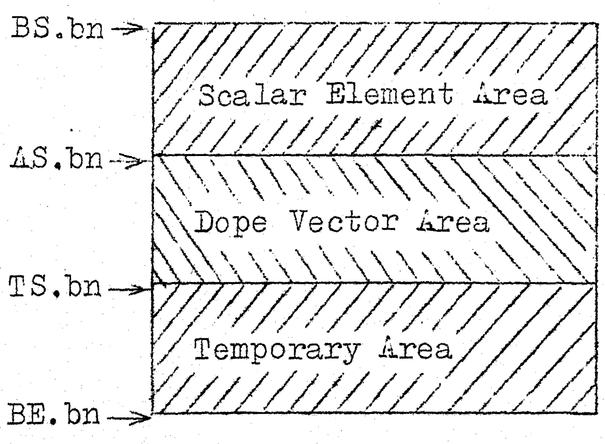
TA(bn) represents the temporary area for block 'bn'

Some interesting conclusions can be drawn from the above exercise.

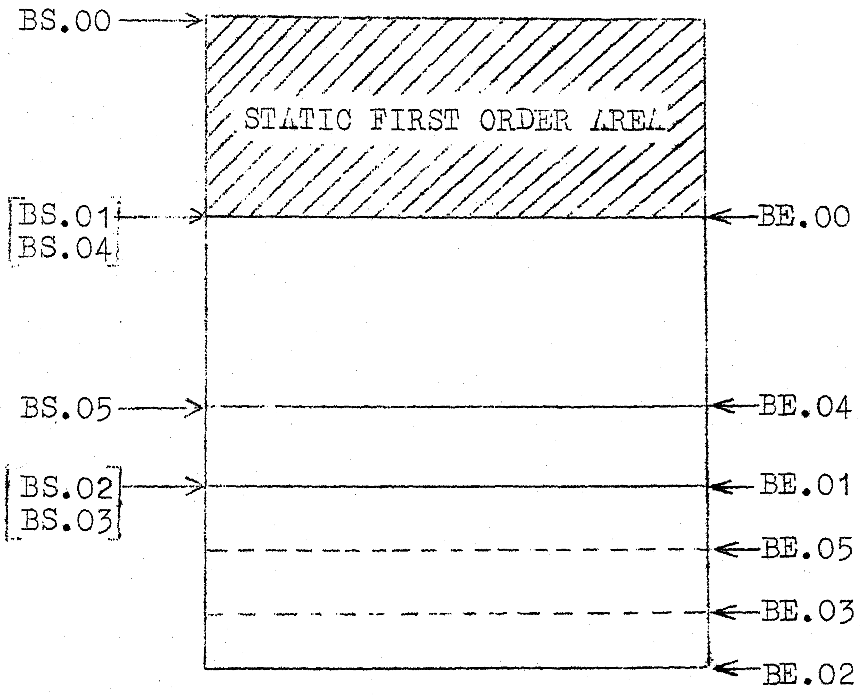
1) Starting addresses of blocks 2(INT1) and 3(INT2) and blocks 1(EXT1) and 4(EXT2) are same. This shows that they share the same first order storage area. This is because of parallelism in their definition. In fact any two blocks having same predecessor would share the same first order area.

ii) Storage assigned to block '0' (imaginary) corresponds to the STATIC variables.

d. <u>Segments of First Order Storage for a Block</u>:



e. <u>Schematic diagram of I order storage assignment for the program considered</u>:



APPENDIX G

G-1 Attributes of Files:

The following shows the attributes that may be assigned to a file as part of a DECLARE statement.

options	Default
INPUT,OUTPUT,UPDATE,PRINT	INPUT
STREAM,RECORD	STREAM
BUFFERED,UNBUFFERED	BUFFERED
SEQUENTIAL	

The file-attributes set is completed as follows:

If Declaration Specifies	PL/7044 will also Assign
UPDATE	RECORD
SEQUENTIAL	RECORD
PRINT	OUTPUT,STREAM
BUFFERED or UNBUFFERED	RECORD, SEQUENTIAL

G-2 Default Attributes:

Storage and scope:

All level one variables: AUTOMATIC is the default for INTERNAL level-one variables; STATIC is the default for EXTERNAL level-one variables. If neither the storage class nor the scope is specified, AUTOMATIC and INTERNAL are assumed.

Arithmetic data: If no attribute is specified for a variable, default attributes are assigned depending upon the first character (or the only character) of the variables. Integer (27 bits)is taken if the first character happens to be I-N. Floating point of single precision is assumed for all the remaining.

String data: If no length specification is specified for a string variable, it is taken as 1.

APPENDIX H

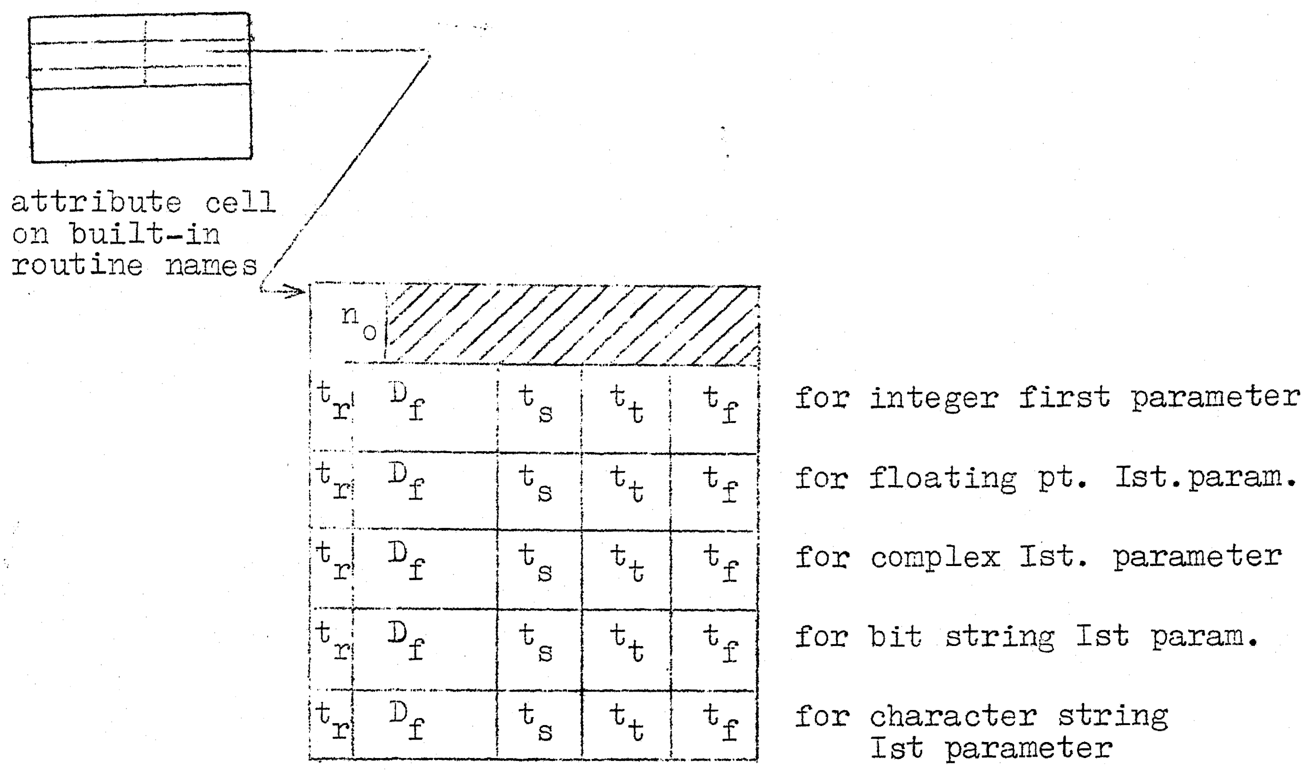
Builtin Procedures and Pseudo Variables:

Because of the generic nature of all the three types of built-in routines (simple built-in, array handling built-in and pseudo variable) of PL/I, symbol table associates an extra six word cell with the attribute cell of builtin name, for holding the description of the generic family members. Selection among these generic members is performed depending on the type of the first parameter of the built-in routine reference. The second, the third, the fourth, the fifth and the sixth word of the associated cell hold the description of generic member for integer, floating point, complex, bit string and character string type of first parameter. Figure APNG-1 illustrates this organisation.

Coding of the type specification (i.e. of t_r,t_s,t_t and t_f) for generic family members varies among the three types of built-in routines. For each type of the built-in routine, the coding of t_r,t_s and t_f is described below.

a) Coding of t_r, t_s, t_t and t_f for built-in routine:

t_r=1	when the returned result is of integer type.
=2	when the returned result is of floating point type.
=3	when the returned result is of complex type.
=4	when the returned result is of bit string (always non-varying) type.
=5	when the returned result is of character string (always non-varying) type.



n_o	▨	▨	▨	▨	
t_r	D_f	t_s	t_t	t_f	for integer first parameter
t_r	D_f	t_s	t_t	t_f	for floating pt. Ist.param.
t_r	D_f	t_s	t_t	t_f	for complex Ist. parameter
t_r	D_f	t_s	t_t	t_f	for bit string Ist param.
t_r	D_f	t_s	t_t	t_f	for character string Ist parameter

<u>The associated extra six word cell</u>

where,

n_o	=	number of parameters associated with this built-in routine.
t_r	=	type of the result returned by a member of the generic family.
D_f	=	Identification number of the entry point for a ~~x~~member of the generic family.
t_s	=	type in which second parameter is expected.
t_t	=	type in which the third parameter is expected.
t_f	=	type in which the fourth parameter is expected

Note: No built-in routines can have more than four parameters. But in case there are more than four parameters supplied in any built-in routine call, all parameters from fifth onwards will be converted, in accordance with the specification for the fourth parameter.

Fig. <u>APNG-1</u>

t_s, t_t, t_f = 0 when any type of argument is allowed in that parameter position.

= 1 when only integer argument is allowed in that parameter position.

= 2 when only floating point argument is allowed in that parameter position.

= 3 when only complex argument can be passed on as parameter.

= 4 when only bit string (any of varying or non-varying type) can be passed on as parameter.

= 5 when only character string (any of varying or non-varying type) can be passed on as parameter.

= 6 when only **literal** integer can appear in that parameter position.

= 7 Either of bit string or character string type of argument can be passed on as parameter. For any other type of argument conversion to character string, is implied.

b) Coding of t_r, t_s, t_t and t_f for array handling built-in routine:

Coding of t_r for array handling built-in routines is identical to that for built in routines of ~~casex~~ (a).

t_s, t_t, t_f = 0 when any type of argument is allowed in that parameter position.

= 1,2,3,4,5 for legal appearance of only scalar integer, floating point, complex, bit string and character string arguments in the parameter position.

= 6,7,8,9,10 for legal appearance of only integer, floating point, complex, bit string and character string arrays in that parameter position.

c) Organisation of attribute cell for a pseudo variable and coding of t_r,t_s,t_t and t_f for pseudo variable generic family members:

Since there exists, always, a built-in function having the same name as pseudo variable, the attribute cell for any pseudo variable name contains a pointer to the attribute cell of the built-in function of same name. Buffer handler chooses (depending on the context) the correct alternative and for built-in function of the same name it follows that pointer to get the corresponding attribute cell address. Organisation of the symbol table entry, corresponding to a pseudo variable name is illustrated in Fig. APNG-2.

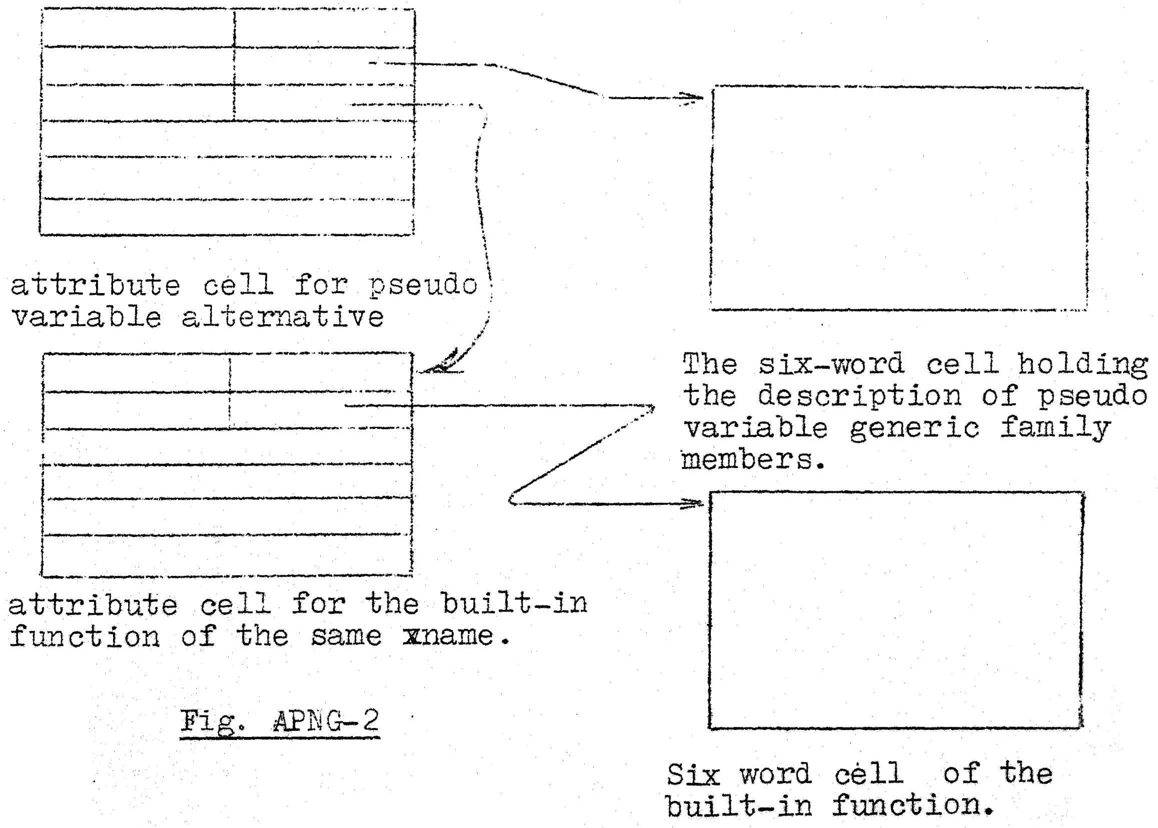

attribute cell for pseudo variable alternative
The six-word cell holding the description of pseudo variable generic family members.
attribute cell for the built-in function of the same ~~x~~name.
Six word cell of the built-in function.

Fig. APNG-2

Coding of t_r, D_f, t_s, t_t and t_f for built in function alternative is identical to that for any other built, in function (case a). For pseudo variable alternative,

t_r = 1 for pseudo variable requiring complex result of the right hand side of '=' expression.

= 2 for pseudo variable requiring bit string (varying or non-varying)result of right hand side of '=' expression

= 3 for pseudo variable requiring character string (varying or non-varying)result of right hand side of '=' expression.

t_s, t_t, t_f = 0 when every operand is allowed in that parameter position.

= 1,2,3,4,5 when integer, floating point, complex, bit string (varying or non-varying) and character string (varying or non-varying) variable (and not expression) is allowed in that parameter position.

= 6,7,8,9,10 when integer, floating point, complex, bit string (varying or non-varying) and character string (varying or non-varying) operand (either variable or expression) is allowed in that parameter position.

APPENDIX I

Statements, Attributes and Restrictions over PL/I:

I-1 Statements:

Following are the alphabetic lists of statements allowed and statements not allowed in PL 7044.

ALLOWED	NOT ALLOWED
BEGIN	ALLOCATE
CALL	DEFAULT
CLOSE	DELAY
DECLARE	DELETE
DISPLAY	FETCH
DO	FLOW
ELSE	FREE
END	LOCATE
ENTRY	NOFLOW
EXIT	ON
FORMAT	RELEASE
GET	REVERT
GOTO	REWRITE
GO TO	SIGNAL
HALT	UNLOCK
IF	WAIT
OPEN	
PROCEDURE	
PUT	
READ	
RETURN	
STOP	
WRITE	

I-2: Attributes:

Attributes have been divided in four major classes. Each class is further sub-divided in various minor classes.

1. Storage and Scope
    - A. Storage
    - B. Scope
2. Data
    - A. Arithmetic
    - B. String
    - C. Label
    - D. Entry
    - E. File
    - F. Task and program control
3. Miscellaneous
4. No-action

The digits and the letters (if any) appended to attributes in the following alphabetic lists of attributes allowed and attributes not allowed, refer to the major class and minor class respectively.

ALLOWED	NOT ALLOWED
ABNORMAL (4)	AREA (2-F)
ALIGNED (4)	BACKWARDS (2-E)
AUTOMATIC (1-A)	BASED (1-A)
BINARY (2-A)	CELL (1-A)
BIT (2-B)	CONTROLLED (1-A)

BUFFERED (2-E)

BUILTIN (2-D)

CHARACTER (2-B)

COMPLEX (2-A)

DECIMAL (2-A)

DIMENSION (3)

ENTRY (2-D)

EXTERNAL (1-B)

FILE (2-E)

FIXED (2-A)

FLOAT (2-A)

GENERIC (2-D)

INITIAL (3)

INPUT (2-E)

INTERNAL (1-B)

IRREDUCIBLE (2-D)

LABEL (2-C)

Length (2-B)

LIKE (3)

MAP (2-D)

NORMAL (4)

OUTPUT (2-E)

Precision (2-A)

PRINT (2-E)

REAL (2-A)

DEFINED (3)

DIRECT (2-E)

ENVIRONMENT (2-E)

EVENT (2-F)

EXCLUSIVE (2-E)

KEYED (2-E)

OFFSET (2-F)

POINTER (2-F)

PICTURE (2-B)

PACKED (4)

POSITION (3)

TASK (2-F)

RECORD (2-E)

Record-size (2-E)

REDUCIBLE (2-D)

RETURNS (2-D)

SECONDARY (1-A)

SEQUENTIAL (2-E)

SETS (2-D)

STATIC (1-A)

STREAM (2-E)

UNBUFFERED (2-E)

UPDATE (2-E)

USES (2-D)

VARYING (2-B)

I.3: Restrictions:

a) Logical Restrictions:

1. Arrays of structures are not allowed.

2. Minor/major structure name should not be used in a GET/PUT DATA statement.

3. Aggregate expression can not be used as an argument in a procedure reference.

4. Recursive function reference should not be made from inside dimension attribute, length attribute, iteration factor or format statement.

5. All DECLARE statements for a block should be placed immediately after the header statement (PROCEDURE/BEGIN). All specifications for parameters of a secondary entry point should be put immediately after the ENTRY statement. No declaration can be made after an ENTRY statement.

6. DO loops can not be used within a GET/PUT DATA statement.

7. Character/Bit strings can not be used within a list or edit or data directed I/O command.

8. A data list must always be specified with a GET DATA command.

9. When a file name is to be transmitted as an argument in a procedure reference, it must either be opened by a OPEN statement or must be used in I/O command lexicographically before its occurrence as argument.

10 A procedure block can not be defined inside a DO group.

b) Storage Restrictions:

1. A statement can be punched any where between column 1 to column 72 (both inclusive)on a 80-column card. Last eight columns may be used for identification.

2. Maximum length of identifiers and character string constants is 30 characters.

3. Maximum length of a bit string constant is 36 bits.

4. Largest absolute value of an integer is $2^{27}-1$ (163708927) (27 bits).

5. A floating point constant should satisfy the relationship $10^{-38} \leq |\text{fp. constant}| \leq 10^{38}$.

6. More than 62 blocks can not be used in the program.

7. More than 63 DO-groups (excluding those used in I/O statements) can not be used.

8. More than 63 structures can not be used.

9. More than 4095 IF statements can not be used.

10. More than 8 labels can not be prefixed to a statement.

11. More than five dimensions in an array can not be used.

12. More than fifteen parameters can not be used while defining a PROCEDURE.

13. More than five structures can not take part in an aggregate expression. Further none of the structures taking part in the aggregate expression should have more than 25 base elements. Structures taking part in a structure expression with BY NAME option specified, should not have more than 10 items at level L+1 under an item at level L.

14. Maximum number of files corresponds to number of utility units available in the installation.

15. A statement can be 500 output-units long at the most. The table for lexical-units and number of output-units is as follows:

Lexical-unit	No. of Output Units
Operators	1
Integers floating point constant, identifier and key words	2
Complex and bit string constants	3
Character string constants	$\lceil n/6 \rceil + 2$

where, n is the length of character string constant.

APPENDIX J

J-1 Program Listing:

Program listing is generated during the first edition. Besides the card-image, it also carries three identification numbers viz. statement number, card number and the block number.

Statement number is used in second edition to refer to the corresponding statement if an error is detected in compilation.

Block number indicates the pairing of the 'END' and the block header statements (PROCEDURE/BEGIN).

The first pass processor prints the type of the END (as to whether closing a DO-group, BEGIN-block or PROCEDURE-block) also. In case of multiple closure besides one such message for each group/block closed, it also prints the message 'MULTIPLE CLOSURE ENCOUNTERED'.

The information about generation of a dummy ELSE is also conveyed to the programmer. The message to this effect appears misplaced on the listing. It should actually appear before the statement detecting the missing ELSE, whereas it appears after that statement. It could have been taken care of but at the cost of increased complexity, which is not warranted if one is familiar to this convention.

The above three facilities help the programmer to keep track of his logic. This is especially helpful in cases like shuffling of cards etc.

At the successful completion of an edition, a message is printed to this effect which shows the progress made by the compiler.

J-2 Errors and the Action Taken:

Complete error message is not printed for the compile time errors. A 6 character mnemonic (3 alphabetic letters followed by 2 digits, 'XXX-nn') uniquely represents an 'error'. However for errors detected at run time, complete error message is printed.

All the errors detected in the first edition appear inter leaved in the program listing. An error detected during the scanning of a statement appears immediately after the card-image of the card on which the error has been detected. However, if the error is detected after the scanning phase is over, the error code follows the card-image on which the statement ends.

At the end of an edition, if it is found that an error has been detected in this edition, all further editions are deleted and and the job is terminated. When an error is detected in a statement, it is immediately skipped and no further analysis is done for that statement.

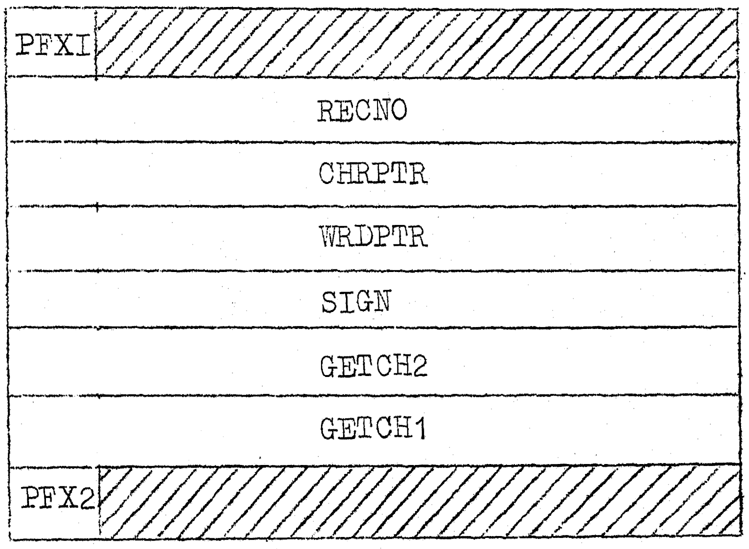
The error messages of the second pass carry the statement number of the statement in error so that proper referencing can be done, since they appear after the program listing.

APPENDIX K

Format of the File Status blocks associated with each stream file:

All file status blocks are 8 words in size.

(a) When file is an input stream file,



where,

RECNO: Record Number = no. of records processed from this file.

CHRPTR: Character pointer - gives the number of character still to be processed in the word pointed to by GETCH1.

WRDPTR: No. of words still to be processed in the current data record.

GETCH2: -ve if a new record is to be read.
+ve if a new record is not to be read.

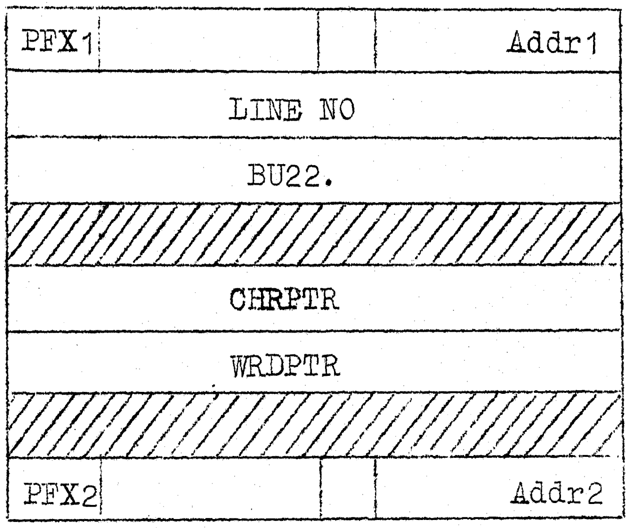
SIGN: -ve for system input file +ve for others.

GETCH1 = Address of the word in the current record being processed.

PFX1 = PZE for stream print files
= PON for stream input
= MZE for record input
= MON for record output

PFX2 = PZE if file has not been used
= MZE if file has been used

b) When the file is an output stream file:



where,

Addr1 = link address of the next file status block
= 0 if this is the last link
$\neq$ 0 if it is not the last link

Addr2 = address of file control block of the file corresponding to this file status block.

LINENO = number of lines or records written in this file.

BU22.	=	pending word in this file.
CHRPTR	=	number of character available in the pending word.
WRDPTR	=	address of word in buffer in complemented from where the next output word is to be placed.

PFX1 and PFX2 have same meanings as in (a).

APPENDIX L

L-1: Buffer handler output for an Assignment Statement:

The coding of the assignment statement A(I,*,J,*),C= I+S+J+T+C1+1+2.0+C2+2+2I+A(*,K1,*,K2)+A(I,*,J,*)+(B*,K1,*)+C(*,*)+ D(1,2); where I, J, D, K1 and K2 have been declared as integer, B, S and T have been declared as floating point and A, C1 and C2 have been declared to be complex,by buffer handler is given below:

Notation:

catpx in the buffer hardler output stands for the complemented address of the attribute cell of the Identifier X.

Buffer handler output in Octal	Comment
0 catpa 0 00032	Operand type of 32_8 denotes the following of array subscripts for address calculation.
0 catpi 0 00001	Operand type of 1 stands for scalar identifier for which normal (and not indirect) address calculation is to be done.
4 00004 0 00044	Operator ',' seperating array subscripts.
0 catpj 0 00001	
4 00004 0 00044	
4 00004 0 00053	Operator ')' denoting end of array subscripts in an array cross-section reference.
0 00000 0 00002	Number of non '*' subscripts.
0 20400 0 00002	Representative word of array cross section.
4 00004 0 00021	First ',' seperating multiple assignment arguments.
0 catpc 0 00033	Operand type of 33_8 denotes array without any subscript list reference.

4 00011 0 00006	
0 catpb 0 00032	Starting of coding of B(*,K2,*)
0 catpk$_2$0 00001	
4 00004 0 00044	
4 00004 0 00053	
0 00000 0 00001	
0 10300 0 00002	
4 00011 0 00006	
0 catpd 0 00032	Starting of coding for D(1,2).
0 00000 0 00003	
0 00000 0 00001	
4 00004 0 00044	
0 00000 0 00003	
0 00000 0 00002	
4 00004 0 00044	
4 00004 0 00051	')' denoting end of subscript list for array element reference.
4 00001 0 00027	Representation of operator devoting end-of-expression. This will be inserted by the calling routine before calling expression processor.

<u>L-2</u>: <u>Expression Processor Output (Before Sorting)</u>:

For buffer handler's coding of the assignment statement of Appendix-L1, the unsorted output of Expression Processor in output buffer and the corresponding entries made in the summary record area are shown below:

<u>Notations</u>:

(1) . Locations of output buffer are addressed symbolically by the symbols attached with the locations. Putting a star on top of those symbols denote 2's complement of those location addresses. (e.g. Sadd1* means 2's complement of the address of location Sadd1).

(ii) Block number and offset of addressing for an identifier X is represented as bx and offsx respectively.

Contents of Output Buffer

Symbolic address	Content of that location in octal	Comment
Sadd1	0 00000 0 00006	Record header for the recursive address calculation output record of A(I,*,J,*).
	0 00101 0 00113	Opcode 113_8 is for generating recursive address calculation output. 01001_8 in decrement is the label where address of array elements will be filled in.
	0 offsa 0 001ba	Address of array header A(in expression processor's output form).
	0 20400 0 00002	Cross section representative word of A(I,*,J,*).
	0 00000 0 00002	Number of non '*' subscripts.
	0 offsi 0 101bi	Subscript I
	0 offsj 0 101bj	Subscript J
Sadd2	0 00000 0 00004	
	0 00102 0 00113	Output for recursive address calculation of c.
	0 offsc 0 001bc	
	0 10203 0 40502	
	0 00000 0 00000	
Sadd3	0 00000 0 00006	
	0 00103 0 00113	
	0 Offsa 0 001ba	Output for recursive address calculation of A(*,K1,*,K2).
	0 10300 0 00002	
	0 00000 0 00002	
	0 offsk$_1$0 101bk$_1$	
	0 offsk$_2$0 1016k$_2$	
Sadd4	0 00000 0 00006	
	0 00104 0 00113	
	0 offsa 0 001ba	Output for recursive address calculation of A(I,*,J,*).
	0 20400 0 00002	
	0 00000 0 00002	
	0 offsi 0 101bi	
	0 offsj 0 101bj	

Saad5	0 00000 0 00005	
	0 00105 0 00113	
	0 offsb 0 001bb	Output for recursive address calculation of B(*,K2,*).
	0 10300 0 00002	
	0 00000 0 00001	
	0 offsk$_2$0 101bk$_2$	
Saad6	0 00000 0 00004	
	0 00106 0 00113	
	0 offsc 0 001bc	Output for recursive address calculation of C(*,*).
	0 10203 0 40502	
	0 00000 0 00000	
Sadd7	0 00000 0 00005	
	0 00107 0 00112	
	0 offsd 0 001bd	Output for address calculation of D(1,2)
	0 00000 0 00002	
	0 00014 0 10500	
	0 00015 0 10500	
Sadd8	0 00000 0 00032	Record header for scalar output.
	0 00001 0 00012	12_8 is opcode for integer addition. 00001_8 is the temporary which will contain the result of this operation.
	0 offsi 0 101bi	
	0 offsj 0 101bj	Negative sign with this operand shows it is bared.
	0 00002 0 00012	
	0 00001 0 11300	Temporary 1
	0 00011 0 10500	Literal 1
	0 00003 0 00012	
	0 00002 0 11300	
	0 01007 0 10700	D(1,2)
	0 00004 0 00071	71_8 is opcode for converting integer operand into floating point.
	0 00003 0 11300	
	0 00005 0 00013	Opcode for floating point addition.
	0 00004 0 21300	
	0 offss 0 201bs	
	0 00006 0 00013	
	0 00005 0 21300	
	0 offst 0 202bt	Indirect addressing is involved with T.

```
          0 00007 0 00014     14₈ is opcode for floating point and
                              complex addition.
          0 00006 0 21300
          0 offsc₁0 301bc₁
          0 00010 0 00015     15₈ is opcode for complex addition.
          0 00007 0 31300
          0 offsc₂0.301bc₂
          0 00011 0 00015
          0 00010 0 31300
          0 00013 0 30500     Literal 2+2I
Saad9     0 00000 0 00022     Record header for repetitive output.
          0 00012 0 00014
          0 01005 0 20700     B(*,K2,*)
          0 00011 0 31300
          0 00013 0 00015
          0 00012 0 31300
          0 01003 0 30700     A(*,K1,*,K2)
          0 00014 0 00015
          0 00013 0 31300
          0 01004 0 30700     A(I,*,J,*)
          0 00015 0 00015
          0 00014 0 31300
          0 01006 0 30700     C(*,*)
          0 00000 0 00056     56₈ is opcode for two word complex
                              assignment.
          0 01002 0 30700     C
          0 00015 0 31300
          0 00000 0 00056
          0 01001 0 30700     A(I,*,J,*)
          0 00015 0 31300
Sadd10    0 00000 0 00012
          0 00010 0 00130     130₈ opcode for bound checking of
                              arrays. 00010₈ denotes serial no.
                              of the recursive address calculation.
```

	0 00000 0 00010	Number of words associated with this operator.
	0 offsa 0 001ba	
	0 20400 0 00002	
	0 offsc 0 001bc	
	0 10203 0 40502	
	0 offsa 0 001ba	
	0 10300 0 00002	
	0 offsb 0 001bb	
	0 10300 0 00002	
Sadd11	0 00000 0 00001	
	0 00000 0 00132	132_8 is opcode denoting starting of repetitive codes.
Sadd12	0 00000 0 00003	
	0 00010 0 00133	133_8 is opcode for closing of the repetitive loops.
	0 00000 0 00134	Opcode 134_8 denotes the end of expression processor output.

Entries in Summary Recorded Area

Content of summary record area in octal	Comments
0 00002 0 00000	Constant entry for collecting all outputs for pseudo variable calls. Zero in the address field denotes existance of no such output.
0 00003 0 Sadd9*	Constant entry for collecting all repetitive output.
0 00004 0 00000	Constant entry for collecting all iSUB defined array repetitive address calculation output.
0 00011 0 Sadd8*	Constant entry for collecting all scalar output.
0 00006 0 Sadd1*	Entry for recursive address calculation of A(I,*,J,*).

0 00006 0 Sadd2*	For C.
0 00006 0 Sadd3*	For A(*,K1,*,K2).
0 00006 0 Sadd4*	For A(I,*,J,*).
0 00006 0 Sadd5*	For B(*,K2,*).
0 00006 0 Sadd6*	For C(*,*).
0 00012 0 Sadd7*	For D(1,2) address calculation.
0 00010 0 Sadd10*	For array bound check orders.
0 00005 0 Sadd11*	For output denoting the starting of repetitive codes.
0 00001 0 Sadd12*	For output denoting the closing of the repetitive loops.

L-3: <u>Expression Processor Output (After Sorting)</u>:

On sorting of expression processor's output (as shown in Appendix L-2) by the sorting routine, Final output, as written in output file, is given below:

Final sorted output in Octal	Comment
7 77777 7 77777	These two words signify the necessity of calling expression processor's code generating routine in third pass.
0 00000 0 00031	
0 01007 0 00112	
0 offsd 0 001bd	
0 00000 0 00002	Output for D(1,2).
0 00014 0 10500	
0 00015 0 10500	
0 00001 0 00012	
0 offsi 0 101bi	
4 offsj 0 101bj	
0 00002 0 00012	
0 00001 0 11300	
0 00011 0 10500	
0 00003 0 00012	Scalar output
0 00002 0 11300	
0 01007 0 10700	
0 00004 0 00071	
0 00003 0 11300	
0 00005 0 00013	
0 00004 0 21300	

```
0 offst 0 202bt
0 00007 0 00014
0 00006 0 21300
0 offsc$_1$0 301bc$_1$
0 00001 0 00015          Scalar output
0 00007 0 31300
0 offsc$_2$0 301bc$_2$
0 00011 0 00015
0 00010 0 31300
0 00013 0 30500

0 00010 0 00130
0 00000 0 00010
0 offsa 0 001ba
0 20400 0 00002
0 offsc 0 001bc          All output for the bound checking of the
0 10203 0 40502          arrays.
0 offsa 0 001ba
0 10300 0 00002
0 offsb 0 001bb
0 10300 0 00002

0 00101 0 00113
0 offsa 0 001ba
0 20400 0 00002          Output for recursive address calculation
0 00000 0 00002          of A (I,*,J,*).
0 offsi 0 101bi
0 offsj 0 101bj

0 00102 0 00113
0 offsc 0 001bc          Output for recursive address calculation
0 10203 0 40502          of C.
0 00000 0 00000

0 00103 0 00113
0 offsa 0 001ba
0 10300 0 00002          Output for recursive address calculation
0 00000 0 00002          of A(*,K1,*,K2).
0 offsk$_1$0 101bk$_1$
0 offsk$_2$0 101bk$_2$

0 00104 0 00113
0 offsa 0 001ba
0 20400 0 00002          Output for recursive address calculation
0 00000 0 00002          of A(I,*,J,*).
0 offsi 0 101bi
0 offsj 0 101bj

0 00105 0 00113
0 offsb 0 001ba          Output for recursive address calculation of
0 10300 0 00002          B(*,K2,*).
0 00000 0 00001
0 offsk$_2$0 101bk$_2$
```

```
       USE  JUNK             Starting of codes for this particular
                             assignment statement.
L01001 PZE  L00014           Subscript list for D(1,2) address
       PZE  L00015           calculation.
       USE  CODE
       TSX  ADDCAL,4
       PZE  L01001           Codes for address calculation of
       PZE  bd+offsd         D(1,2).
       STA  L00107
       CLA  bi+offsi
       SUB  bj+offsj
       ADD  L00011           Literal 1 is added to the partial
                             result of I+J.
L00107 ADD  **               Adding of D(1,2) to the partial
                             result.
       ORA  =023300000000    Converting partial result from integer
       FAD  =023300000000    to floating point.
       FAD*  bs+offss
       FAD   bt+offst
       FAD   L00012          Addition of literal 2.0 to the partial
                             result.
       FAD   bc1+offsc1
       STO   Temp+0          Temp and Temp+1 are the temporary
                             locations for holding partial result.
       CLA   bc1+offsc1+1
       STO   Temp+1
       CLA   Temp+0
       FAD   bc2+offsc2
       STO   Temp+0          Addition of C2 to the partial result.
       CLA   Temp+1
       FAD   bc2+offsc2+1
       STO   Temp+1
       CLA   Temp+0
       FAD   L00013
       STO   Temp+0          Addition of literal 2+2I with the
       CLA   Temp+1          partial result.
       FAD   L00013+1
       STO   Temp+1
       TSX   SUPLBN,4              Initialise standared locations ior
       PZE   ba+offsa              holding lower bounds and   upper
       OCT   020400000002          bounds, with those of the first
                                   array operand.

       TSX   CHKBND,4
       TRA   L01002
       PZE   3
       PZE   bc+offsc
       OCT   010203040502
       PZE   ba+offsa              Check other arrays (taking part as
       OCT   010300000002          operands) for having lower and upper
       PZE   bb+offsb              bounds equal to those of the first
       OCT   010300000002          array operand.
L01002 EQU   *

       USE   JUNK

L01003 EQU   *                     Subscript list for recursive address
       PZE   bi+offsi              calculation of A(I,*,J,*).
       PZE   bj+offsj

L01004 PZE   L00101                List of addresses where starting
       PZE   0I1010                address and increments  will be
       PZE   0I2010                filled in by RLRCAC routine.

       USE   CODE
       TSX   RLRCAC,4              Output for calling of recursive
       PZE   L01003                address (real) calculation initiali-
       PZE   ba+offsa,,L01004       sation routine for A(I,*,J,*).
       OCT   020400000002

       USE   JUNK

L01005 EQU   *

L01006 PZE   L00102

       PZE   1I1010

       PZE   1I2010

       USE   CODE

       TSX   RLRCAC,4              Calling of RLRCAC routine for
       PZE   L01005                initialisation of starting address
       PZE   bc+offsc,,L01006         and increments for the recursive
       OCT   010203040502             address calculation of C.

       USE   JUNK

L01007 EQU   *
       PZE   bk1+offsk1
       PZE   bk2+offsk2
```

```
L01010 PZE  L00103
       PZE  2I1010
       PZE  2I2010
       USE  CODE
       TSX  RLRCAC,4             For recursive address calculation
       PZE  L01007               of A(*,K1,*,K2).
       PZE  ba+offsa,,L01010
       OCT  010300000002
       USE  JUNK
L01011 EQU  *
       PZE  bi+offsi
       PZE  bj+offsj
L01012 PZE  L00104
       PZE  3I1010
       PZE  3I2010
       USE  CODE
       TSX  RLRCAC,4
       PZE  L01011               For recursive address calcula-
       PZE  ba+offsa,,L01012     tion of  A(I,*,J,*).
       OCT  020400000002
       USE  JUNK
L01013 EQU  *
       PZE  bk2+offsk2
L01014 PZE  L00105
       PZE  4I1010
       PZE  4I2010
       USE  CODE
       TSX  RLRCAC,4             For recursive address calculation
       PZE  L01013               of B(*,K2,*).
       PZE  bb+offsb,,L01014
       OCT  010300000002
       USE  JUNK
L01015 EQU  *
L01016 PZE  L00106
       PZE  5I1010
       PZE  5I2010
       USE  CODE
       TSX  RLRCAC,4
       PZE  L01015               For recursive address
       PZE  bc+offsc,,L01016     calculation of C(*,*).
       OCT  010203040502
       TRA  STC010        Transfer straight to the starting
                          of repetitive codes.
1L0010 CLA  LB1.1         LB1.1 contains  lower bound (first
                          copy) for the first '*' in array
                          cross-section reference.
       ADD  =1
       STO  LB.1
       SUB  UB.1          UB.1 contains the upper bound for the
                          first '*' in array cross-section.
       TZE  100010        Transfer out of this particular loop.
       LXA  L00101,2      Get the next element address for
0I1010 TXI  *+1,2,**      A(I,*,J,*).
       SXA  L00101,2
       LXA  L00102,2
1I1010 TXI  *+1,2,**      For C.
       SXA  L00102,2
       LXA  L00103,2
2I1010 TXI  *+1,2,**      For A(*,K1,*,K2).
       SXA  L00103,2
       LXA  L00104,2
3I1010 TXI  *+1,2,**      For A(I,*,J,*)
       SXA  L00104,2
       LXA  L00105,2
4I1010 TXI  *+1,2,**      For B(*,K2,*)
       SXA  L00105,2
       LXA  L00106,2
5I1010 TXI  *+1,2,**      For C(*,*)
       SXA  L00106,2
       TRA  STC010
2L0010 CLA  LB1.2         LB1.2 contains lower bound (first copy)
                          of the second '*' position in array
                          cross-section reference.
       ADD  =1
       STO  LB1.2
       SUB  UB.2          UB.2 contains corresponding upper
                          bound.
       TZE  200010
       LXA  L0101,2       Getting address of the next
0I2010 TXI  *+1,2,**      element of A(I,*,J,*).
       SXA  L00101,2
       LXA  L00102,2
1I2010 TXI  *+1,2,**        For C.
       SXA  L00102,2

       LXA  L00103,2
2I2010 TXI  *+1,2,**        For A(*,K1,*,K2).
       SXA  L00103,2

       LXA  L00104,2
3I2010 TXI  *+1,2,**        For A(I,*,J,*).
       SXA  L00104,2

       LXA  L00105,2
4I2010 TXI  *+1,2,**        For B(*,K2,*).
       SXA  L00105,2

       LXA  L00106,2
5I2010 TXI  *+1,2,**        For C(*,*).
       SXA  L00106,2

STC010 EQU  *               Starting of repetitive codes.

L00105 CLA  **
       FAD  Temp+0
       STO  Temp+0          Adding of B(*,K2,*) with the partial
       CLA  Temp+1          result.
       STO  Temp+1

       CLA  Temp+0
L00103 FAD  **
       STO  Temp+0          Adding of A(*,K1,*,K2) with the
       LAC  L00103,2        partial result.
       CLA  Temp+1
       FAD  1,2
       STO  Temp+1

       CLA  Temp+0
L00104 FAD  **
       STO  Temp+0
       LAC  L00104,2        Adding of A(I,*,J,*) with the partial
       CLA  Temp+1          result.
       FAD  1,2
       STO  Temp+1

       CLA  Temp+0
L00102 STO  **              Assigning the result of the
       LAC  L00102,2        expression to C.
       CLA  Temp+1
       STO  1,2

       CLA  Temp+0
L00101 STO  **
       LAC  L00101,2        Assigning the  result of the expression
       CLA  Temp+1          to A(I,*,J,*).
       STO  1,2
       TRA  2L0010      Repeat the loop for next array element
                        in the  cross-sections.
200010 CLA  LB2.2       LB2.2 contains the second copy of
                        lower bound for the second '*' in
                        array cross-section reference.
       STO  LB1.1
       TRA  1L0010      Repeat the loop for next cross-
                        section (i.e. for the next subscript
                        value of the first '*').
100010 EQU  *
```

Note: Even though for bound checking only one of the two references of A(I,*,J,*) was supplied, for recursive address calculation the two references were treated to be different.

APPENDIX M

M-1: Coding of an Edit Directed Output Statement with Nested 'DO' Groups:

The statement:

```
PUT (FILE1) SKIP (<exp1>) EDIT(((A,B,<exp2> DO
                               2 1
I=1 TO M BY N WHILE C>O), P, ( (Q,R,S DO J=N TO L1 BY L2
                       1'    4 3
WHILE  CDASH<O)  DO K=MM TO NK BY KK WHILE K2<O) ,T, W
              3'                                4'
DO L(3) = J,K, 1 TO 10 BY 2) ) (<immediate format item>);
                           2'
```

In the above statements the first order storage addresses are as given below:

A ...	BS.01+000001	L1 ...	BS.01+000012
B ...	BS.01+000002	L2 ...	BS.01+000013
C ...	BS.01+000003	CDASH	BS.01+000014
M ...	BS.01+000004	K ...	BS.01+000015
N ...	BS.01+000005	KK ...	BS.01+000016
P ...	BS.01+000006	T ...	BS.01+000017
Q ...	BS.01+000007	W ...	BS.01+000018
R ...	BS.01+000008	MM ...	BS.01+000019
S ...	BS.01+000009	NK ...	BS.01+000020
I ...	BS.01+000010	K2 ...	BS.01+000021
J ...	BS.01+000011		

We assume that B,P,Q and R are formal parameters and the remaining an ordinary variables. Further W is an array of dimension 3. Those variables starting with I,J,K,L,M and N are integers while the rest are floating point numbers.

```
       The Map Code                      Comments

       TSX     .IOSUP,4
       PON     PLF.00,,F.00      Open file and validate file
       EXTERN  .IOSUP            operation.

     arithmetic code for
        exp1

       TSX     IOHSC.,4          Execute SKIP command.
L00001 PZE     **
       EXTERN  IOHSC.
       TRA     M00001            G  · for performing format linkage.
M00002 EQU     *
       TRA     M00004            Transfer to coding of 'DO'Group 2-2'.
M00006 TRA     **                DOMAIN of 'DO' Group 1-1' begins.
       AXT     BS,01+000001,1
       PXA     ,1
       TSL     HNLIO.            Output variable A.
       EXTERN  HNLIO.
       CLA     BS.04+000002
       TSL     HNLIO.            Output variable B.

    arithmetic code for
          exp2

L00002 AXT     **,1
       PXA     ,1
       TSL     HNLIO.            Output expression.
       TRA     M00006            DOMAIN OF 'DO',Group 1-1' ends.
M00005 TRA     **                DOMAIN of 'DO' Group 2-2' begins.
       CLA     =1                Coding of 'DO' Group 1-1' begins.
       STO     BS.01+000010      Initialize 'DO' index.
       CLA     BS.01+000004
       STO    TS.01+000001    Save upper limit in temporary.
       CLA    BS.01+000005
       STO    TS.01+000002    Save increment in temporary
X00001 CLA    TS.01+000001    check if 'DO' index has crossed
                              the limit.
       SUB    BS.01+000010
       TMI    Y00001

     arithmetic coding for while
     clause. Requires two labels.
     TXXXXX  and FXXXXX

TXXXXX TSL    M00006          Call to Domain of 'DO' Group 1-1'
       CLA    BS.01+000010    increment 'DO' index.
       ADD    TS.01+000002
       STO    BS.01+000010
       TRA    X00001
FXXXXX EQU    *
Y00005 EQU    *               Coding of 'DO' Group 1-1' ends.
       CLA    BS.01+000006
       TSL    HNLIO.          Output variable P.
       TRA    M00007          Transfer to coding of 'DO' Group 4-4'.
M00011 TRA    **              Domain of 'DO' Group 3-3' begins.
       CLA    BS.01+000007
       TSL    HNLIO.          Output variable Q.
       CLA    BS.01+000008
       TSL    HNLIO.          Output variable R.
       AXT    BS.01+000007,1
       PXA    ,1
       TSL    HNLIO.          Output variable S.
       TRA    M00011          Domain of 'DO' Group 3-3' ends.
M00010 TRA    **              Domain of 'DO' Group 4-4' begins.
       CLA    BS.01+000005    Coding of 'DO' Group 3-3' begins
       STO    BS.01+000011    initialize 'DO' index.
       CLA    BS.01+000012
       STO    TS.01+000001    Save upper limit.
       CLA    BS.01+000013
       STO    TS.01+000002    Save increment.
```

```
X00002 CLA     TS.01+000001    Check if 'DO' index has crossed
       SUB     BS.01+00011     the upper limit.
       TMI     Y00002
    arithmetic code for while
    clause.  Requires the
    labels
       TYYYYY and F YYYYY

TYYYYY TSL     M00011          Call domain of 'DO' Group
       CLA     BS.01+000011
       ADD     TS.01+000002    Increment index.
       STO     BS.01+000011
       TRA     X00002
FYYYYY EQU     *
Y00002 EQU     *
       TRA     M00010          Coding of 'DO' group 3-3' and
                               Domain of 'DO' group 4-4' end.
M00007 EQU     *
       CLA     BS.01+000019
       STO     BS.01+000015    Initialize 'DO' index.
       CLA     BS.01+000020
       STO     TS.01+000001    Save upper limit
       CLA     BS.01+000016
       STO     TS.01+000002    Save increment
X00003 CLA     TS.01+000001    Check if 'DO' index has
                               crossed up per limit.
       SUB     BS.01+000015
       TMI     Y00003

    arithmetic coding of while
    clause; requires two labels
    TZZZZZ  and  FZZZZZ

TZZZZZ TSL     M00010          Call domain of 'DO' Group 4-4'.
       CLA     BS.01+000015
       ADD     TS.01+000002    Increment index.
       STO     BS.01+000015
       TRA     X00003
FZZZZZ EQU     *
```

```
Y00003 EQU     *               Coding of 'DO' Group 4-4' ends.
       AXT     BS.01+000017,1
       PXA     ,1
       TSL     HNLIO.          Output variable T.
       TSX     F.RNGE,4        To calculate the  starting address,
       PZE     BS.01+00018,,   increment and range for array W.
       ETC     Z00004
       PZE     Z00005,,Z00006
Z00004 AXT     **,1
       PXA     ,1
       TSL     HNLIO.
Z00005 TXI     *+1,1,**
       SXA     Z00004,1
Z00006 TXL     Z00004,1,**
       TRA     M00005          Domain of 'DO' Group 2-2' ends.
M00004 EQU     *               Coding of 'DO' Group 2-2' begins.

    arithmetic coding for
    calculating address
    of L(3)

L00003 AXT     **,1
       SXA     TS.01+000001,1  Store address of 'DO' index.
       CLA     BS.01+000011
       STO*    TS.01+000001    Initialize 'DO' index.
       TSL     M00005          Call Domain of 'DO' Group 2-2'.
       CLA     BS.01+000015
       STO*    TS.01+000001    Initialize 'DO' index.
       TSL     M000005         Call Domain of 'DO' Group 2-2'.
       CLA     =1
       STO*    TS.01+000001    Initialize 'DO' index.
       CLA     =10
       STO     TS.01+000002    Save upper limit.
       CLA     =2
       STO     TS.01+000003    Save increment.
```

```
X00007 CLA     TS.01+000002   Check if 'DO' index has
       SUB*    TS.01+000001   crossed the limit.
       TMI     Y00007
       TSL     M00005         Call domain of 'DO' Group 2-2'.
       CLA*    TS.01+000001
       ADD     TS.01+000003
       STO*    TS.01+000001
       TRA     X00007
X00007 EQU     *
       TRA     M00003         Get out of output statement.
M00001 TSX     STHIO.,4       call format link age routine.
       PZE     M00012
       TRA     M00002         Go to beginning of Edit Statement
M00012 EQU     *

     format list

M00003 EQU     *
       .
       .
       .
```

M-2: Coding of a DO statement outside an I/O command:

The statement:

```
DO I = 1,2,3, 10 TO 20 BY 2, M TO N BY K
WHILE B<0;
```

First order address of I is BS.01+000001, of M is BS.01+000002, of N is BS.01+000003, of K is BS.01+000004.

Let the DO serial number be 2.

```
CLA     =2             Update do-serial number.
STO     DOSRNO         Update DO-serial number.
CLA     =1
STO     BS.01+000001   Initialize 'DO' index
TSL     C00002         Call  domain of DO.
```

```
        CLA      =10
        STO      BS.01+000001   Initialize 'DO' index.
        CLA      =20
        STO      TS.01+000001   Save upper limit in temporary.
        CLA      =2
        STO      TS.01+000002   Save increment in temporary
X00001  CLA      TS.01+000001   check if 'DO' index exceeds the
        SUB      BS.01+000001   upper limit.
        TMI      Y00001
        TSL      C00002         Call domain of 'DO' Group.
        CLA      BS.01+000001
        ADD      TS.01+000002   Increment 'DO' index and loop back.
        STO      BS.01+000001
        TRA      X00001
Y00001  EQU      *
        CLA      BS.01+000002   Initialize 'DO' index.
        STO      BS.01+000001
        CLA      BS.01+000003
        STO      TS.01+000001   Save upper limit in temporary.
        CLA      BS.01+000004
        STO      TS.01+000002   Save increment in temporary.
X00002  CLA      TS.01+000001   Check if 'DO' index has exceeded
        SUB      BS.01+000001   limit.
        TMI      Y00002

     while clause coding,
     requires labels
     TXXXXX  and  FXXXXX

TXXXXX  TSL      C00002
        CLA      BS.01+000001   Increment 'DO' index,and loop.
        ADD      TS.01+000002
        STO      BS.01+000001
        TRA      X00002
FXXXXX  EQU      *
Y00002  EQU      *
        TRA      P00002         Getout of 'DO' group.
```

```
C00002 TRA      *

    Body of the 'DO' Group

       TRA      C00002          These two instructions are
                                generated when 'end' of 'DO'
P00002 EQU      *               Group is recognised.
        .
        .
        .
```

M-3: Input and Output Formats for Stream Oriented Data Transmission in PL 7044:

List Directed Input:

Data in the input stream has one of the following general forms:

a) +|- arithmetic constant.

b) Character string constant.

c) Bit string constant.

d) +|- real constant +|- imaginary constant.

An arithmetic constant may be fixed point (integer) or floating point constant and may have binary or decimal representation. Data in the data stream will be called source and the variable to which the data is to be assigned will be called target.

Data items in the stream must be separated by a blank or by a comma. The separator may be preceded and/or followed by an arbitrary number of blanks. A null field in the stream is indicated either by the very first non-blank character in the stream being a comma or by two adjacent commas separated by an arbitrary number of blanks. The transmission of the list directed input is terminated by expiration of the data list or the end-of-data or end-of-file condition. In the latter case, the processing of the job will also be terminated.

If the data is a character string constant, the surrounding quotation marks are deleted and double quotation marks within the string are treated as single quotation marks. Character strings may not have a length greater than 128. Bit strings may not have a length greater than 36.

If data is an arithmetic constant or a complex constant, it is converted to coded arithmetic form. The target attributes are then examined and source conversion if necessary, is performed. It is then assigned to the target. The accompanying table gives the possible conversions.

List Directed Output:

The value of the scalar variable is converted to character representation and transmitted to the data stream. Two blanks always procede the transmission of a data value. If the length of a numeric data item is longer than the number of character positions remaining in the current line (record), the entire item is outputted starting at the beginning of the next record. Character strings whose length exceeds the size of a record will be split between two or more records. No blanks will be inserted when such an item is continued on the next line.

All fixed point arithmetic data items are outputted in I(11) format. All floating point arithmetic data items are printed in E(14,8) format. All complex data items are printed in 2E(14,8) format. For character and bit strings, the format is set according to their present length.

<u>Data-Directed Input</u>:

The data directed data in the input stream has the following format:

Scalar-variable = Constant ȸ[, scalar variable = Constant] ...... $

Scalar variable may be subscripted name with optionally signed decimal integer constants for subscripts.

Each assignment may be separated by an arbitrary number of blanks or a comma.

The constant has the same format as in the case of list-directed input.

The scalar variable names used on the left hand side of the '=' sign must be one of the names used in the data list used in the corresponding GET DATA statement. The order in which names appear in the data-list need not be same as the order in which they appear in the stream nor is it necessary for all the names in the data list to appear in the data stream.

The data list may not include major or minor structures. Structure elements may be used, however, provided the number of characters forming the base element including the dot between qualifying names, does not exceed thirty.

<u>Data Directed Output</u>:

The general format for data directed output is the following:

Scalar variable = Constant ƀƀ Scalar variable
ƀ=ƀ constant ...

If an array occurs in the data list as a data element, then the elements with their subscripts are transmitted in row major order.

Each assignment is separated by two blanks. The constant has the same format as in the case of list- directed output.

Structures and 'DO' repetitive groups are not allowed.

The length of the data field in both input and output is a function of the length of the identifier and that of the associated data constant. If, during output transmission, the length of the data field exceeds the number of remaining character positions in the record (line) then a new line is started and the data item is then outputted on this line.

<u>Edit Directed Input and Output</u>:

The edit directed I/O format is governed by the associated format specifications whose syntax and semantics are given in the chapter on 'Analysis of Format Statements'.

TABLE OF CONVERSION FOR LIST DIRECTED INPUT

TARGET ↓ / SOURCE →	INTEGER	FL. POINT	COMPLEX	BIT ST.	CHAR ST.	BINARY INTEGER	BINARY FL. POINT
INTEGER	✓	✓	✓ REAL PART ONLY	✓	✓ ONLY NUMERICS	✓	✓
FLOATING POINT	✓	✓	✓ REAL PART ONLY	✓	✓ ONLY NUMERICS & '.', E, ±	✓	✓
COMPLEX	✓	✓	✓	✓	✓ ONLY NUMERICS & '.', E, ±	✓	✓
BIT STRING	✓	X	X	✓	✓ ONLY 0's & 1's	✓	X
CHAR STRING	✓	✓	✓	✓	✓	✓	✓

✓ : CONVERSION POSSIBLE . X : CONVERSION NOT POSSIBLE

APPENDIX N

Output of Format Generator:

Input:

```
(5((M)(X(10), COLUMN(M), C(F(10,M),E(M,<exp>)),
 (M)I(M),'RESULT', A(<exp>), B(3),5Ø(12)))).
```

The MAP code generated will be as follows:

```
            AXT         5,1
            TSL         IOHLP.
            EXTERN      IOHLP.

            CLA*        BS.02+000100
            AXT         0,1
            TMI         *+2
            PAX         ,1
            TSL         IOHLP.

            TSX         IOHXC.,4
            PZE         10
            EXTERN      IOHXC.

            CLA*        BS.02+000100
            TPL         *+2
            ZAC
            STA         *+2
            TSX         IOHCC.,4
            PZE         **
            EXTERN      IOHCC.

            CLA*        BS.02+000100
            AXT         0,1
            TMI         *+2
            PXA         ,4
            SXD         *+2,4
            TSX         IOHFC.,4
            PZE         10,,**
            EXTERN      IOHFC.

        [arithmetic coding]

            CLA*        BS.02+000100
            AXT         0,1
            TMI         *+2
            PAX         ,1
            SXA         *+2,1
            TSX         IOHEC.
     L00001 PZE
            EXTERN      IOHEC.
        CLA*      BS.02+000100
        AXT       0,2
        TMI       *+2
        PAX       ,2

        CLA*      BS.02+000100
        TPL       *+2
        ZAC
        STA       *+2
        TSX       IOHIC.,4
        PZE       **
        EXTERN    IOHIC.,4
        PZE       **
        EXTERN    IOHIC.
        PZE
        TSX       IOHHC.,4
        PZE       6
        BCI       1,RESULT
        EXTERN    IOHHC.

      [arithmetic  coding]

        TSX       IOHAC.,4
L00002  PZE
        EXTERN    IOHAC.

        TSX       IOHBC.,4
        PZE       3
        EXTERN    IOHBC.

        AXT       5,2

        TSX       IOHOC.,4
        PZE       12
        EXTERN    IOHOC.

        TSL       IOHRP.
        EXTERN    IOHRP.

        TSL       IOHRP.

        TRA       IOHEF.
```

APPENDIX Ø

Run Time Stack Management Routines (RTSMR):

a. **List of RTSMR:**

1.	Open a begin block	R.OPBG
2.	Invoke a procedure block	R.OPPR
3.	Terminate a block by 'END' Statement	CLSEND
4.	Terminate a block by RETURN statement.	CLPRWT
5.	Terminate a block by RETURN statement and return a value.	CLPRRT
6.	GOTO statement with block closing.	GTBLOK.

b. **Link Cell Format:**

▨	L.P.P.	▨	ret addr
▨	W.A.P.	▨	# Saved
▨	DOSRNO	▨	Calling block no.

where,

P.P.P. (Present Procedure Pointer): This is a one word area which contains the address of the first of the 3 locations which constitute the linkage cell of the block which is being currently executed.

L.P.P. (Last Procedure Pointer): This is the value of the P.P.P. when the invocation of the present block was made.

W.A.P. (Work Area Pointer): It is the address of the first free location after arrays and character variables (Second order storage area) have been allocated memory space. This denotes the address of the first free cell available for assigning temporaries requiring more than 2 words.

F.C.P. (Free Cell Pointer): It is the address of the first free location after multiword temporaries have been assigned.

Saved: It is the number of first order locations saved, if any, when the present block was invoked.

ret addr (return address): It is the address of the calling point, if the invoked block is a PROCEDURE block. It is zero for a BEGIN block. The return pointer is needed at the time of terminating the block.

Calling block no.: It is the serial number of the calling block.

DOSRNO: It is the serial number of the innermost DO-group open at the calling point, if any.

PSRNO.: Global location which contains the serial number of the innermost block open.

Notation:

(LNKCEL, i) i=0,1,2 represents the (i+1)-st word of the linkage cell.

A	is used to represent the address portion.
D	is used to represent the decrement portion.
LNKCEL	is same as F.C.P.

1. R.OPBG:

Calling sequence:

```
        AXT     blkopn,1
        TSX     R.OPBG,4
```

where,

'blkopn' is the block number of the block to be opened.

Actions Taken:

```
BLKOPN        ←XR1     index register '1'
D(LNKCEL,0)   ←P.P.P.
A(LNKCEL,0)   ←0
P.P.P.        ←F.C.P.
F.C.P.        ←F.C.P.+3
D(LNKCEL,1)   ←W.A.P.
A(LNKCEL,1)   ←0
D(LNKCEL,2)   ←DOSRNO
A(LNKCEL,2)   ←PSRNO.
PSRNO.        ←BLKOPN
```

(Some of the pointers are being saved unnecessarily because it shares the routine with R.OPPR.)

Return of Control:

```
TRA   1,4
```

2. R.OPPR:

Calling Sequence:

```
        TSX  R.OPPR,4
        TXI  *+3,,arglst
        PZE  PHWadd
        PZE  tempad,, mnrtyp
```

for procedure invocation by a CALL statement the last word is

```
        PZE  0
```

where,

'arglst'	is the address of the argument list, if any.
'PHWadd'	is the address of the procedure header word.
'tempad'	is the address of the result receiving temporary
'mnrtyp'	is the minor type of the result returned.

Actions Taken:

1. Extract the number of the block to be opened from the PHW and, store it in BLKOPN.

2. Determine if any saving is to be done.

   $BS\langle blkopn\rangle - BE.\langle psrno\rangle \geq 0 \Rightarrow$ no saving

   $< 0 \Rightarrow$ saving is to be done

   If saving is done, update S.A.P. accordingly.

   $\#$ saved $= |BS\langle blkopn\rangle - BE.\langle psrno\rangle|$

   S.A.P. $\leftarrow$ S.A.P. + $\#$ Saved.

3. D(LNKCEL,0) $\leftarrow$ P.P.P.

   A(LNKCEL,0) $\leftarrow$ XR4 Contents of Index Register 4.

P.P.P. $\leftarrow$ F.C.P.

F.C.P. $\leftarrow$ F.C.P.+3

D(LNKCEL,1) $\leftarrow$ W.A.P.

D(LNKCEL,1) $\leftarrow$ # Saved   if saving done.

$\leftarrow$ 0   if no saving is done.

D(LNKCEL,2) $\leftarrow$ DOSRNO

A(LNKCEL,2) $\leftarrow$ PSRNO.

4. All DO-groups open at CALL point, if any, are closed.

5. Do prologue initialisation, if 'prglst' $\neq$ 0.

6. PSRNO. $\leftarrow$ BLKOPN

Return of Control:

```
CLA*  2,4
STA   *+1
TRA   **
```

3. **CLSEND:**

Calling Sequence:

```
TSX  CLSEND,4
```

Actions Taken:

1. If sign of (LNKCEL,0) is -ve, it implies the main procedure is being closed. Program is terminated.

2. A(LNKCEL,0) = 0 $\Longrightarrow$ terminate a BEGIN block.

$\neq$ 0 $\Longrightarrow$ terminate a PROCEDURE block

2.1 For terminating a BEGIN block:

(Entry point CLEND1)

F.C.P. $\leftarrow$ P.P.P.

P.P.P. $\leftarrow$ D(LNKCEL,0)

W.A.P. $\leftarrow$ D(LNKCEL,1)

PSRNO. $\leftarrow$ A(LNKCEL,2)

DOSRNO $\leftarrow$ D(LNKCEL,2)

Return:

```
        TRA  1,4
```

2.2 For Terminating a PROCEDURE block:

(Entry point CLEND.)

a. A(LNKCEL,1) = 0, $\Longrightarrow$ no restoring to be done

$\neq$ 0 $\Longrightarrow$ restoring first order storage area to be done.

Start address of the area to be restored is BS.⟨PSRNO.⟩ It is determined from the storage table. The restoration starts from S.A.P. and goes to S.A.P. - # saved.

At the end of restoration, update S.A.P. as

S.A.P. $\leftarrow$ S.A.P. - # saved

b. DOSRNO $\leftarrow$ D(LNKCEL,2)

If DOSRNC $\neq$ 0, the DO-groups that were closed at the time of invoking this procedure are opened once again i.e. their do-status bits are put ON.

XR 4 $\leftarrow$ A(LNKCEL,0)

c. If (3,4) = 0, no result is expected

$\neq$ 0, result is expected, hence error.

```
d.              F.C.P.        <-P.P.P.
                P.P.P.        <-D(LNKCEL,0)
                W.A.P.        <-D(LNKCEL,1)
                PSRNO.        <-A(LNKCEL,2)
```

Return:

```
                TRA   1,4
```

4. CLPRWR:

Calling sequence:

```
                    TSX      CLPRWR,4
```

Actions taken:

```
LOOP    CLA*      P.P.P.
        TMI       Error      Trying to close the main
                             procedure by a RETURN statement.
        PAX       ,4
        TXH       CLEND.,4,0
        AXT       1-LOOP,4
        TRA       CLEND1
```

CLEND. and CLEND1 are two entry points in the CLEND routine, which skip some of the unwanted code. It can be seen from the present scheme, that all the BEGIN blocks, those are open, also get closed. This process is repeated till a procedure block invocation is obtained.

5. CLPRRT:

Calling sequence

```
                TSX     CLPRRT,4
                PZE     tempadd,,mnrtyp
```

a. First it determines the result receiving temporary, if any, and the minor type of result expected. Now it finds out whether any conversion is to be done. After doing the necessary conversion, value returned is assigned to result temporary.

b. Now it merges with CLPRWR with the difference that error clause in 3.b gets nullified.

6. GTBLOK:

Calling sequence

```
        TSX     GTBLOK,4
        PZE     label
        PZE     sblno,, dblno
```

where,

'sblno' is the source block number.

'dblno' is the destination block number.

Actions:

Close block by calling CLSEND routine at the CLEND entry point till the destination block number is obtained.

Return:

```
        TRA*  1,4
```

APPENDIX P

FREE LIST:

Free core area, in second pass, has been divided into six word cells. These cells are needed for multiword temporary storage by a number of routines viz. expression processor, 'BY NAME' option analysis routine, symbol table etc. To realize economy in storage space, a linked list is maintained from which cells can be taken when needed and to which cells could be returned when no more needed. This ensures that no cell is lost;

Since symbol table makes maximum use of the free list, the size of a symbol table cell fixed the size of the attribute cell.

The stack-discipline of the free list is last in first out (LIFO). A cell which is returned to the free list is added to one end of the free list. The cell which was last added would be the first to be given on a call for a new cell.

Two pointers (FRCELL, LSCELL) are maintained which store the address of the cells which is to go out first/to which the cell being returned would be chained and which is at the other end of the chain respectively. In case the latter cell is taken, list becomes empty.

The format of the free list is shown in Fig. P-1.

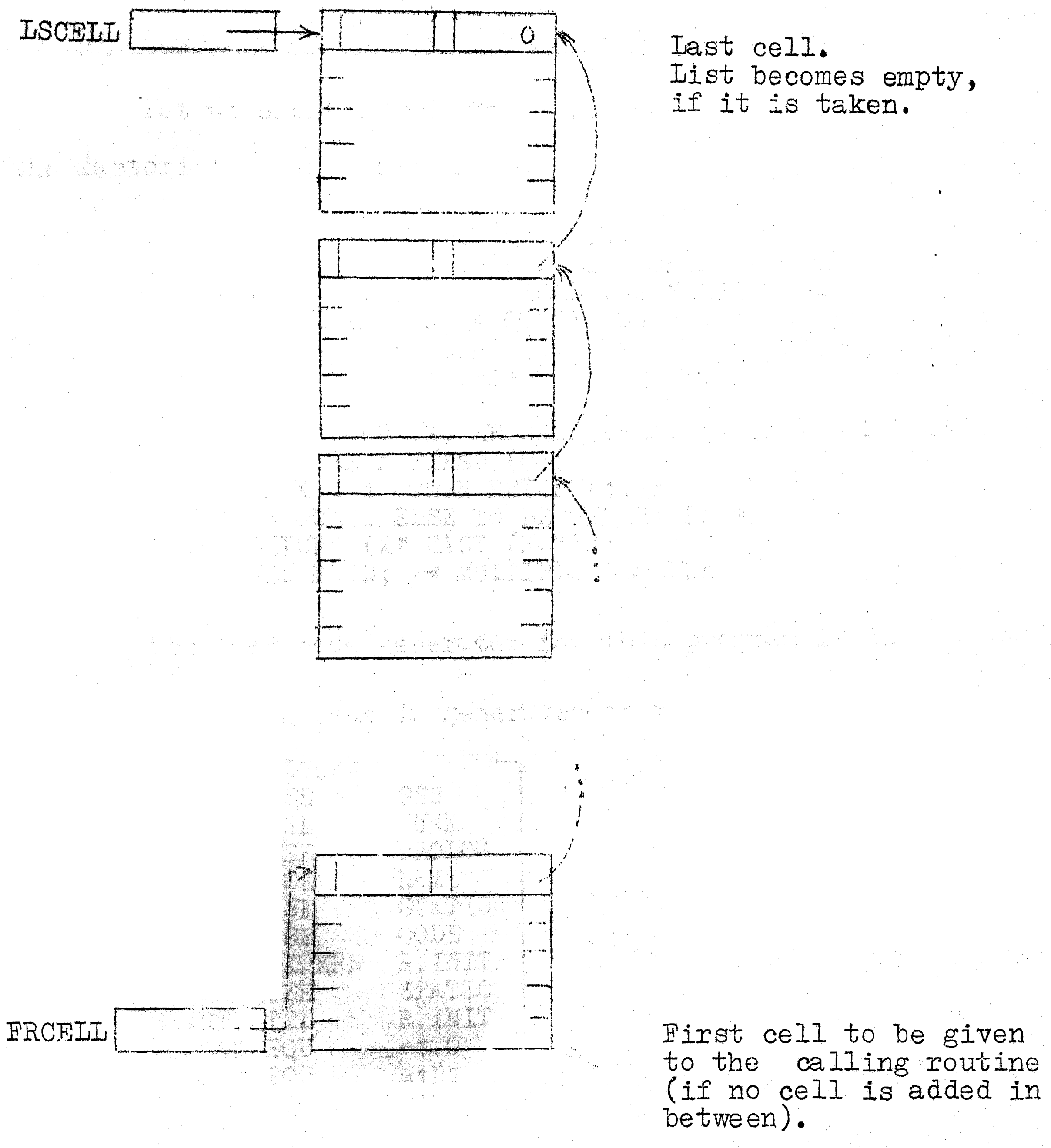


Fig. P-1: Free List.

APPENDIX Q

PL 7044 Compiler Output for a PL/I Program:

Let us consider the following PL/I program for computing the factorial of an integer.

```
MAIN: PROCEDURE OPTIONS (MAIN);
      /* RECURSIVE COMPUTATION OF FACTORIAL */
      GET LIST (M); /* READ THE VALUE OF M */
      PUT EDIT ((N, FACT(N)  DO N = 1 TO M))
               (COLUMN (1) 'FACTORIAL OF'I(4)'='
               E(14,8));

FACT: PROCEDURE (X) RETURNS (REAL (10,1)) RECURSIVE;
      DECLARE X FIXED (8);
      IF X = 1  THEN RETURN(1.);
      /* DUMMY ELSE TO BE GENERATED */
      RETURN (X* FACT (X-1));
      END MAIN; /* MULTIPLE CLOSURE */
```

The MAP code generated for this program is as follows:

Following code is generated in the second pass

```
$IBMAP PL7044
        USE     BSS
        USE     JUNK
        USE     PROLOG
        USE     NAME                  Common to every PL/I program.
        USE     STATIC
        USE     CODE
        EXTERN  R.INIT
        USE     STATIC
START   TSL     R.INIT
L00000  EQU     =1.0
L00001  EQU     =1B1
L00002  EQU     =1

        USE     JUNK
L00005  VFD     Ø3/1,15/000000,Ø3/0,15/L00003   PHW for MAIN
L00011  VFD     Ø3/2,15/L00010,Ø3/0,15/L00007   PHW for FACT
        USE     PROLOG
L00010  EQU     *          Prolog list for FACT
        VFD     3/0,15/BS.02+000000,6/1,6/7,6/5
        PZE     0          End of prologue list
L00013  EQU     =Ø201400000000    Literal 1.0
L00014  EQU     =Ø201400000000    Literal 1.0
L00015  EQU     =Ø000000000001    Literal 1
```

Following code is generated in the third pass.

```
L00003 EQU    *          Entry point for MAIN
       TSL    DCL.01
       TRA    DCE.01
DCL.01 PZE    **         Closed procedure for declarations
       TRA*   DCL.01     of block MAIN.
DCE.01 EQU    *
L00006 EQU    *
       USE    JUNK
S.FSBN PON    0          File status block for system
       BSS    6          input file
       PZE    0
       USE    CODE
       TSX    .IOSUP,4   Call to I/O supervisor
       PZE    S.FBIN,,S.FSBN
       EXTERN .IOSUP
       TSX    G.LIST,4   Call to list directed inputting
                         routine.
       PZE    BS.01+000000  'address of M.
       EXTERN G.LIST
       USE    JUNK
S.FSBU PZE    0          File status block for system
       BSS    6          output
       PZE    0
       USE    CODE
       TSX    .IOSUP,4   Call to I/O Supervisor
       PON    S.FBOU,,S.FSBU
       TRA    M00000     Go and perform format linkage
M00001 EQU    *
       TRA    M00003     Goto coding of 'DO'.
M00004 TRA    **         Domain of 'DO' starts.
       AXT    BS.01+000001,1
       PXA    ,1
       TSL    HNLIO.     Output N
       EXTERN HNLIO.
       USE    JUNK
L00017 EQU    *          Argument list for function call FACT
       PZE    BS.01+000001
       USE    CODE
       EXTERN R.OPPR
       TSX    R.OPPR,4   Call to function FACT
       TXI    *+3,,L00017
       PZE    L00011
       PZE    TS.01+000000,,2
       AXT    TS.01+000000,1
       PXA    ,1
       TSL    HNLIO.     Output FACT(N)
       TRA    M00004     Eng of Domain of 'DO'.
```

```
M00003 EQU    *             Coding of 'DO'.
       CLA    =1
       STO    BS.01+000001        Initialise index
       CLA    BS.01+000000
       STO    TS.01+000001        Save upper limit.
       CLA    =1
       STO    TS.01+000002        Save increment.
X00000 CLA    TS.01+000001        Check if index has
       SUB    BS.01+000001        crossed limit.
       TMI    Y00000              Go out of 'DO' loop
       TSL    M00004              Transfer to Domain
       CLA    BS.01+000001
       ADD    TS.01+000002
       STO    BS.01+000001
       TRA    X00000
Y00000 EQU    *
       TRA    M00002
M00000 EQU    *                   Format list starts.
       TSX    TSHIO.,4
       PZE    M00005
       TRA    M00001
       EXTERN TSHIO.
M00005 EQU    *
       TSX    IOHCC.,4
       PZE    1
       EXTERN IOHCC.
       TSX    IOHEC.,4
       PZE    13
       BCI    3, FACTORIAL OF
       EXTERN IOHHC.
       TSX    IOHIC.,4
       PZE    4
       EXTERN IOHIC.
       TSX    IOHHC.,4
       PZE    1
       BCI    1,=
       TSX    IOHEC.,4
       PZE    14,8
       EXTERN IOHEC.
       TRA    IOHEF.              Format list ends.
       EXTERN IOHEF.
M00002 EQU    *
       TRA    END.02              Tra around procedure FACT.
L00007 EQU    *                   Entry point for FACT.
       TSL    DCL.02
       TRA    DCE.02
DCL.02 PZE    **                  Closed procedure for
       TRA*   DCL.02              declarations of FACT.
DCE.02 EQU    *
L00012 EQU    *
       CLA*   BS.02+000000        Coding for IF statement
                                  starts.
```

```
INTFLT MACRO                             Macro for integer
       ORA       =Ø233000000000          to floating point
       FAD       =Ø233000000000          conversion
       ENDM
       INTFLT
       CAS       L00013                  Literal 1.0
       TRA       F00016
       TRA       S00016
       TRA       F00016
S00016 EQU       S.0001
F00016 EQU       F.0001
S.0001 EQU       *
       EXTERN    CLPRRT
       TSX       CLPRRT,4                Code for RETURN (1.)
       PZE       L00014,,2               Literal 1.
       TRA       E.0001
F.0001 EQU       *
E.0001 EQU       *
       CLA*      BS.02+000000
       SUB       L00015                  Literal 1
       STO       TS.02+000000
       USE       JUNK
L00020 EQU       *                       Argument list for function
                                         call FACT (X-1).
       PZE       TS.02+000000
       USE       CODE
       TSX       R.OPPR,4
       TXI       *+3,,L00020             Call to FACT
       PZE       L00011
       PZE       TS.02+000000,,2
       CLA*      BS.02+000000
       INTFLT
       STO       TS.02+000001
       LDQ       TS.02+000001
       FMP       TS.02+000000
       STO       TS.02+000000
       TSX       CLPRRT,4                Code for RETURN(X*FACT(X-1)
       PZE       TS.02+000000,,2
       EXTERN    CLSEND
       TSX       CLSEND,4                END for FACT
END.02 EQU       *
       TSX       CLSEND,4                END for MAIN
END.01 EQU       *
```

```
*       Following code is generated at the end of third pass.

               ENTRY    BLPDTB
        BLPDTB EQU      *                      Block predecessor table
               OCT      000000000000
               OCT      000000000000
               OCT      000000000001
               ENTRY    STGTAB                 Block storage table
        STGTAB EQU      *
               PZE      BE.00,,BS.00
               PZE      BE.01,,BS.01
               PZE      BE.02,,BS.02
        AS.00  EQU      BS.00+000000
        TS.00  EQU      AS.00+000000
        BE.00  EQU      TS.00+000000
        BS.01  EQU      BE.00
        AS.01  EQU      BS.01+000002
        TS.01  EQU      AS.01+000000
        BE.01  EQU      TS.01+000003
        BS.02  EQU      BE.01
        AS.02  EQU      BS.02+000001
        TS.02  EQU      AS.02+000000
        BE.02  EQU      TS.02+000002
        BS.00  EQU      *
               BSS      000008
               ENTRY    DOTBL                  Do predecessor table start.
        DOTBL  EQU      *
               END      START
```

APPENDIX R

Glossary of Important Terms and Words:

Activating a 'BEGIN' block/procedure block:	A BEGIN block is said to be activated when control passes through the BEGIN statement for the block. A procedure block is said to be activated when procedure is invoked at any one of its entry points
Asynchronous Operations:	See Synchronous operations.
Attribute:	An attribute is a descriptive property associated with a name to describe a characteristic of a data item or file which the name may represent.
Base Element:	See structure.
Begin Block:	See Block.
Block:	A block is a collection of statements that defines the program region or scope throughout which an identifier is established as a name. A BEGIN block can be activated only by the normal sequential flow of the program and can be used wherever a statement can be used. A procedure block can only be activated remotely by CALL statements or by function references.
Body of a Block:	The body of a block is defined as the collection of statements which appear lexicographically between the heading statement and the trailing statement of the block.
Compile Time:	This is the period during which a program is being compiled.
Compile Time Data Table:	This is a fixed table of size 500 words used for storing data elements appearing in data directed I/O.

Data List:	A data list is a list of data elements appearing in an I/O statement.
Data Set:	A collection of data external to the program constitutes a data set.
Deactivating a Block:	A block is said to be deactivated when control is transferred to any one of the following statements (a) END (b) STOP (c) RETURN (valid only in case of a procedure), (d) GOTO statement whose destination is a point outside the block.
DO Index:	A scalar variable which is used for controlling the number of times the domain the 'DO' is to be executed is known as the 'DO' index.
DO Parameter/Specification:	A 'DO' specification is defined as $\text{expression -1}\begin{cases}\text{TO expression-2 [BY expression-3]}\\ \text{BY expression-3 [TO expression-2]}\end{cases}$ [WHILE (expression)]
Domain of 'DO':	This is the collection of statements which appear lexicographically between the 'DO' statement and the corresponding END statement.
Dynamic Descendence:	If a block B is active, and another block B1 is activated from a point internal to block B (while B still remains active, then B1 is said to be the immediate dynamic descendent of B. If block B1 has a immediate dynamic descendant in B2, then B2 is said to be the dynamic descendent of B.
Dynamic Encompassing:	If block B is a dynamic descendent of block A, then block A dynamically encompasses block B and block B is dynamically encompassed. by A.
Epilogue:	See prologue.

Expression:	An expression is an algorithm used for computing a value.
	Scalar Expression: A scalar expression is one that has a scalar value.
	Aggregate Expression: An aggregate expression is an expression involving one or more aggregate operands i.e. structure or array.
Execution Time:	This is the period during which the control is with a) the object code generated by the compiler and b) the supporting library routines.
First Order Storage Area:	This is the area where all scalar variables both STATIC and AUTOMATIC , array header word(s), dope vector words and single and double word temporaries.
Field Count:	This is the number of times a format item is to be repeated.
Fixed Table:	A binary search table of all the keywords (or pseudo reserved words), builtin procedure names and pseudo variable names.
Format Explicitor:	The entity defines the field width in case of the following: A-, B-, I-, X-, O-, COLUMN-, LINE-, and SKIP- format items. And in case of E- and F- format items, the first explicitor defines the field width and the second explicitor defines the number of places after the decimal point.
Group Count:	It is the number of times a group of format items have to be repeated.
Implementation Dependency:	There are certain aspects of a language which cannot be rigorously defined without recourse to the characteristics of the machine on which the language is to be implemented. Such features are said to be implementation dependent.

Immediate Predecessor:	See predecessor
Immediate Dynamic Descendant:	See Dynamic Descendant.
Linear Space:	**This is a concept wherein the mapping between the** memory space and the name space is the identity function. The addresses given by the loader in the name space are the same as in the memory space
Lexical Unit:	The program source text is broken up into units having unique codes recognisable by the syntax analysing routines. Such units are called lexical units.
Pass:	A pass is defined as a scan through the text of the program. The text may be the source text or the text prepared by some other pass. During the scan, the text is modified and a new text is produced.
Partitioning of Memory:	In this report, partitioning of memory refers to software division of available core space. In PL 7044, the core space has been divided into the following: Nucleus and IOCS, First Order Storage, Second Order Storage, Third Order Storage, Object Code, Library Routines, and I/O Buffer Area.
Predecessor:	A is said to be the predecessor of B if B appears lexicographically between the heading statement and the trailing statement of A. If in addition, A is closest to B (lexicographically) then A is said to be the immediate predecessor of B.
Primary entry Point:	A primary entry point is that entry point identified by the left most entry on the PROCEDURE Statement. All other names for the procedure identify the secondary points.

Prologue: On entering a block certain initial actions are performed. These initial actions constitute the prologue. Actions that are performed when a block is deactivated constitute the epilogue.

Pseudo Variables:

In general, pseudo variable are built-in functions that can appear wherever other variables can appear in order to receive values. However, in PL 7044 the use of pseudo variables is much more restrictive.

Recursion: Recursion is said to have taken place when a procedure is reactivated while it is still active.

Run Time Stack: The second and third order storage share the free core available after loading the object program in what may be termed a double ended stack. This stack is referred to as the run time stack.

Secondary Entry Point:

See primary entry point.

Second Order Storage:

It is the storage space occupied by the elements of an array, character scalar variables, and multiword (more than 2 words) temporaries.

Structure: A structure is a hierarchical collection of scalar variables, arrays and structures. The outer most structure is called the major structure. All contained structures are called minor structures.

Base Element: Elements of structures which are themselves not structures are known as base elements.

Synchronous Operation:

A program in which the execution of statements proceeds serially in time is said to be executed synchronously.

When several statements are executed in parallel in time, then the operation is said to be asynchronous.

TASK:	A task is an identifiable execution of a set of instructions and exists only during the execution of the set of instructions. Note that TASK is not a set of instructions but the execution of the instructions.

BIBLIOGRAPHY

1. PL/I Language specification - IBM; Order Number GY33-6003-2.

2. IBM System/360, PL/I Reference Manual; Form Number C28-8201-1.

3. A Guide to PL/I - S.V. Pollock and T.D. Sterling-Holt, Reinhart and Winston, 1969.

4. NPL: Highlights of a New Programming Language - George Radin and H. Paul, Rogoway, CACM 8, 1965, p. 9.

5. Automatic Syntactic Analysis - W.M. Foster, MacDonald and Elsevair Inc.

6. Compiling Techniques, F.R.A. Hopgood, MacDonald and Elsevair Inc.

7. Anatomy of a Compiler - John A.N. Lee, Reinhold Publication Group, New York, 1967.

8. Computer Programming and Computer Systems - Anthony Hassit, Academic Press, 1967.

9. ALGOL-60 Implementation - B. Randell and L.J.Russel, Academic Press, 1964.

10. Translation of ALGOL-60 - Grau,Hill and Lambark, Springer-Verlag, New York, 1967.

11. A Multipass Translation Scheme for ALGOL-60, Hawkins and Huxtable, Annual Review in Computer Programming-3, Paragon Press.

12. WATFOR Documentation - University of Waterloo.

13. Programming Systems and Languages - Saul Rosen, McGraw-Hill 1967.

14. Scatter Storage Techniques-Robert Morris, CACM Vol. 11 p.38 (Jan. 1968).

15. Syntactic Analysis and Operator Precedure - R.W.Floyd, JACM-10, 1963, p. 316.

16. An Algorithm for Coding Efficient Arithmatic Operations-R.W.Floyd, CACM4, 1961, p. 42.

17. Sequential Formula Translation , K. Samelson and F.L.Baur, CACM3, 1960, p. 76.

18. High Speed Compilation of Object Code - C.W.Gear,CACM,8, 1965, p. 483.

19. Peephole Optimization by W.M. McKeeman, CACM 8, 1965,p.443.

20. Recursive Processes and Algol Translation, A.A.Grau, CACM4, 1961, p. 10.

21. Bounded Context Translation - Robert M. Graham, MIT, Computer Centre Report.

22. Syntax Directed Compiling, T.E. Cheatham and Sattley K., Proc. of SJCC 1964, Washington D.C.

23. Data Directed Input-Output in FORTRAN-CACM 10,1967,p.35.

24. System Programmers Guide IBM Form No. C-28-6339-5.

25. Input Output Control System IBM Form No. C-29-6309-4.

26. Programmers Guide, IBM Form No. C-28-6318.